

PYTHON 3 TUTORIAL

Language Tutorials, GUI Tutorials, and Projects for Electrical Engineers and Scientists

Rick A. Crist (Retired EE)

CREATED 2023-2024 BY RICK A. CRIST FOR EDUCATIONAL PURPOSES. SELF-PUBLISHED DOCUMENT.

Introduction

As a young electrical engineer, I designed electronic circuits using desktop computers and commercial circuit simulators such as EEsOf, MicrowaveOffice, HP ADS, etc. I was curious how these circuit simulators were programmed.

Python is the language selected for this tutorial and projects. According to [Tiobe Index](#) Python is the most popular programming language as of Oct 2023 and continues to increase in popularity. It has been the language of the year 5 times since 2007. It has all the features and performance needed for Electrical Engineering circuit simulation.

This tutorial will focus on the language features and libraries needed to create a circuit simulator including the following topics:

- Python 3 Tutorial
 - GUI Tutorial using CustomTkinter

- Beginner Project
 - Scientific Calculator
 - Intermediate Projects
 - Shape Editor
 - Line Editor
- Advanced Projects
 - Diagram Editor to draw and manipulate shapes and lines on a GUI canvas
 - Digital Simulator for the design and simulation of digital circuits
 - Analog Simulator for the design and simulation of analog circuits
 - RF/Microwave Simulator for the design and simulation of RF/microwave circuit

The source code is available on GitHub at .

Chapter 1 - Python Basics

This chapter will be a crash course in Python programming including how to install Python and the PyCharm IDE and verify the installations using the Python version of Hello World. We will create the Hello World program using ChatGPT AI as a "programming assistant". We will then cover basic Python coding such as comments, variables, data types, operators, input & output, selection code, functions, looping and iteration, list data structure, classes, objects, modules, and an introduction to graphical user interface (GUI) programming. I recommend that you type all example programs rather than copying my source code. It is like learning a new language, after awhile, your mind will begin speaking the new language fluently or in this case typing fluently.

Python and PyCharm Installation

Many installation tutorials and videos are available so I won't repeat them here. Watch the YouTube video: [How to Install Python 3.11 and PyCharm on Windows](#). Install Python 3.11.5 rather than 3.11.4 as shown in the video. Install the latest version of PyCharm.

It is important to know about installation of earlier Python versions because 3rd Party Libraries tend to lag the release of new Python versions and you may need to install an earlier version to use a Library.

Ensure that your python directory is added to the Path in the Advanced settings. Open a command window and enter python --version. You should see the following if the installation and path are ok.

```
D:\EETools\Python3Tutorial>python --version
Python 3.11.5
```

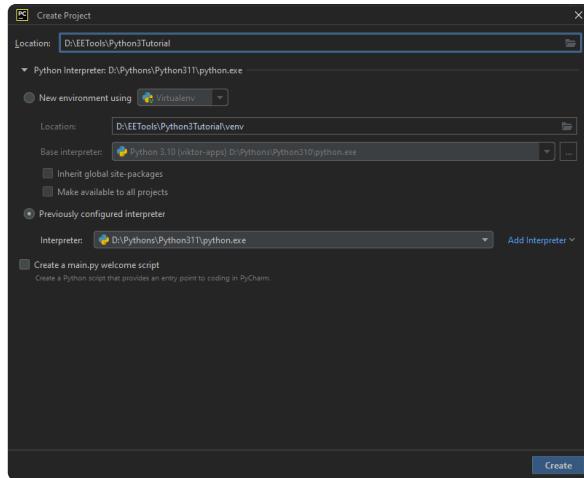
Project Setup

Create a project directory, my directory is D:/EETools. The subdirectories in the location will contain the PyCharm projects for the tutorials. Run PyCharm and create a new project. In the Create Project dialog, set the location to D:\EETools\Python3Tutorial and set the

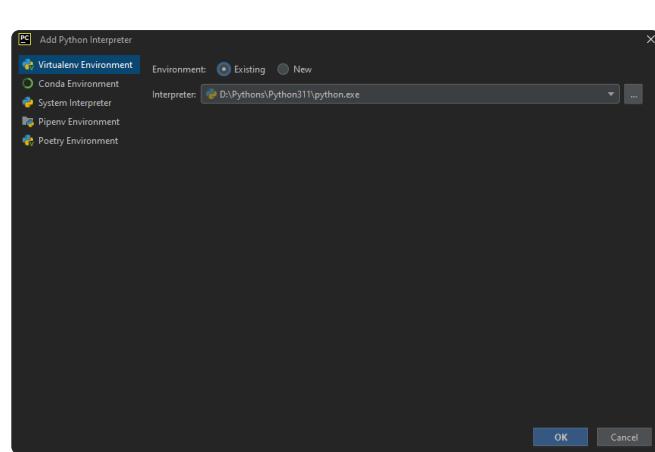
previously configured interpreter to your Python 3.11.5 python.exe file as shown in the video.

One of the best features of PyCharm is that it allows easy configuration of "Virtual Environments" which isolates this project from all other projects allowing you to define the python interpreter and 3rd party libraries to this project.

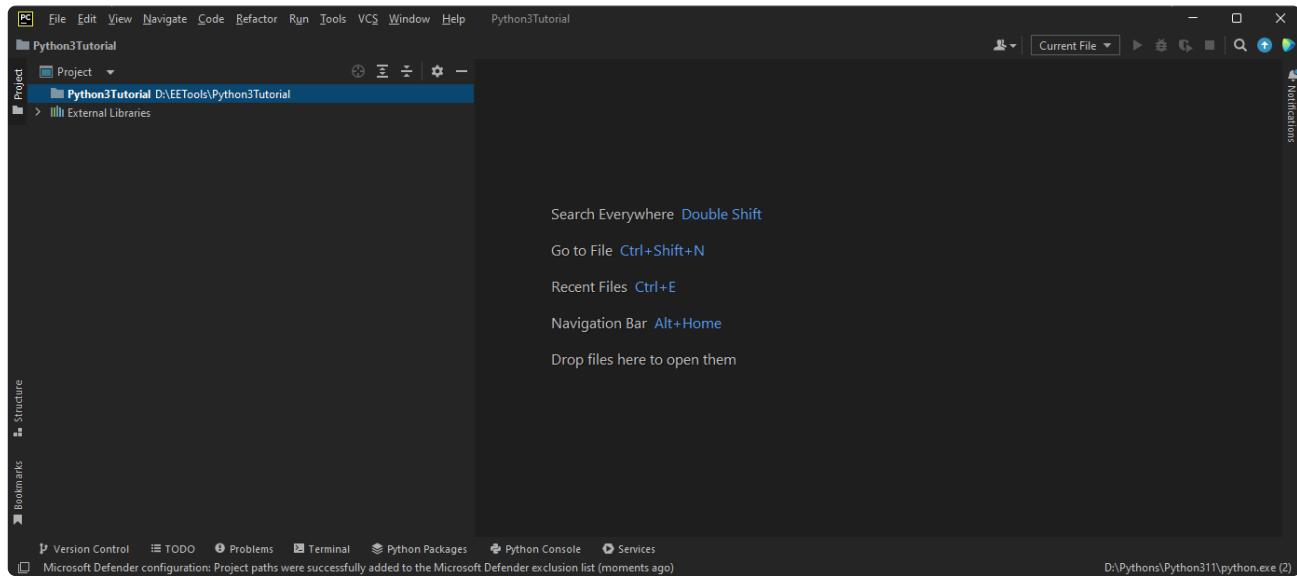
Create Project Dialog



Python Interpreter Dialog



If all goes well, the PyCharm IDE user interface will open.



The current Python version is shown in the lower right corner. The Python directory is shown and the “venv” or virtual environment is listed. Source code is shown in the upper right window. The lower window is used to access the Terminal, Python Console, and run console as we will see after the first program is run.

Hello World

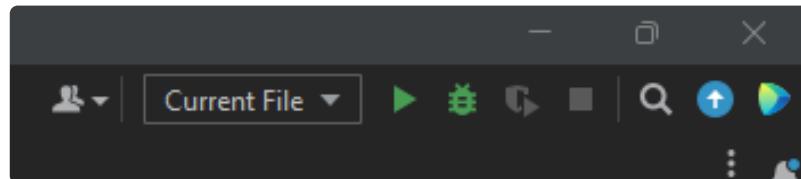
Many python tutorials call the main python program main.py. However, it is not necessary to do that. Create a new python file by selecting File > New > Python File > Name: hello_world. Note that it is not necessary to add the .py extension if you choose Python File. A new python file is created called hello_world.py. You can also right click on the project name: Python Tutorial > New > Python File > Name: hello_world. If you select File rather than Python File you will need to add the .py extension. If you need to change the name of a file, right click on the filename > Refactor > Rename > Name: hello_world.py.

On line 1 in the hello_world.py file enter the following python code:

```
print("Hello World!")
```

Note that PyCharm will verify compliance to PEP8 coding style by issuing warnings when styles are violated. In our simple Hello World program, a blank line is needed after the print line to comply with PEP 8. The style guide can be found at [PEP 8 - Style Guide for Python Code](#). I have found that if I follow the style guide, I am a better programmer.

Check the Select Run/Debug Configuration drop down that it is set to "Current File". Run the program by clicking on the green arrow button.



PyCharm calls the configured interpreter (Python 3.11.5) and produces the output in the console:

```
D:\Pythons\Python311\python.exe D:\EETools\Python3Tutorial\hello_world.py
Hello World!

Process finished with exit code 0
```

If the program runs successfully, it demonstrates that the Python and PyCharm installations are working and we are ready to proceed with the Tutorial.

Let's look at the anatomy of the hello_world.py program.

```
print("Hello World!")
```

`print()` is a built-in function in Python 3 that sends the specified message to the standard output device which in this case is the PyCharm output console. We will learn how and why to define functions later. The message in this case is a string. A string is a sequence of characters enclosed in single or double quotes. We will learn more about strings in a later section.

This may seem like a trivial example but we will use `print()` functions in all programs, even those with a graphical user interface to debug the program. This is accomplished by placing `print()` statements at key locations in the program to display data in order to troubleshoot a problem. We can also use the PyCharm debugger to find problems in the code. We will discuss both debugging methods later.

Pat yourself on the back for completing installation and running your first Python program. Try experimenting with the program by changing the string to print your name in the message.

Artificial Intelligence (AI) Programming Assistant

An AI such as ChatGPT can be used as a programming assistant for small coding challenges. Note that a program generated by AI may contain programming errors or "bugs".

Open [ChatGPT](#) and start a new chat. Note that currently ChatGPT-3.5 is free to use.

Send the following message:

Write a python hello world program.

Default (GPT-3.5)

R Write a python hello world program

Certainly! Here is a simple "Hello, World!" program in Python:

```
python
print("Hello, World!")
```

To run this code, make sure you have Python installed on your system. Copy the code into a file with a '.py' extension (for example, 'hello.py'), and then run the script using a Python interpreter. You'll see the output:

```
Hello, World!
```

ChatGPT correctly creates our `hello_world.py` program and even shows the expected output. To verify the code, copy the program into a PyCharm file and run it. AI is very helpful as a programming assistant to give you ideas on how to write code to perform a specific task. The "Regenerate" function will rerun the request and often shows another method to write the code. This can be much faster than using a search engine to find the same methods. Note that ChatGPT can be configured with a voice interface.

Comments

Comments are messages to a programmer reading the source code. Comments are not executed by the python interpreter. Python supports single-line comments, multi-line comments, and inline comments. Comments can be used by the programmer to "comment out" a line or multiple lines to try new code before deleting old code or to see the effect of removing some code from execution.

Create a new program called `comments.py`.

```
# Single line comments

"""

Multi-line comment
using
triple quotes
"""

print("Python comments are awesome") # This is an inline comment
```

```
# print("This line is commented out")
```

Later we will learn about doc strings which are used to document functions and classes.

Variables & Operators

A variable is a human readable name for data that is stored in computer memory. Without variables, the programmer would need to provide the memory address where the data is stored, not very user friendly.

Variables are "assigned" a value that can be of different "data types" such as:

- integers (number without decimal point)
- floats (number with a decimal point)
- string which we have already encountered
- boolean - true or false
- data containers such as list, dictionary, set, tuple
- custom user types
- many other types

Operators perform "operations" on values and variables called "operands". Types of operators in Python include:

- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Identity Operators
- Membership Operators

Create a new program called `variables_and_operators.py`

```
# Assigning various data types to variables  
  
# Integers
```

```
a = 1
b = 2

# Integer math operators
add = a + b
sub = b - a
mult = a * b
div = a / b

# Print string and values for math operations
print("a + b = ", add, " b - a = ", sub, " a * b = ", mult, " a / b = ", div)
# Print on a single line
print("a + b = ", add, "\nb - a = ", sub, "\na * b = ", mult, "\na / b = ",
div) # Print on multiple lines
```

Console Output

```
a + b = 3  b - a = 1  a * b = 2  a / b = 0.5
a + b = 3
b - a = 1
a * b = 2
a / b = 0.5
```

We will cover other data types and operators as they are needed in the examples. We will also create a scientific calculator later with a graphic user interface (GUI) that uses these operators.

Input and output

User input refers to interaction between the program user (human) and the program (computer). Output refers to computer generated data that is displayed on the computer screen such as the print() function we have already discussed. Although console programs are easy to code and see quick results, most if not all modern programs have a GUI interface. We will discuss GUI programming in detail later.

Create a new program called input_output.py

```
# Get user input
name = input("What is your name?: ")

# Output a message to the computer screen using the provided input
print("Hello ", name)
```

Console Output - Note that I entered my name: Rick

```
What is your name?: Rick
Hello Rick
```

Selection using if...elif...else

Computer programs can make decisions based on certain conditions in the program. The if...elif...else is a python programming statement that allows the program to execute code blocks if conditions are met. All programming languages have a way to define a code block. Many languages use curly brackets to define the block. Python uses indentation to define a code block. The indentation is usually 4 spaces for code readability.

In Python, the simply if statement syntax is:

```
if condition:
    code executed if condition is true
```

Create a new program called selection.py

```
# Define a variable for the ambient temperature
temperature = 12

# Simple if statement
if temperature < 32:
    print("It is cold outside")

# if..else statement
temperature = 75
if temperature < 32:
    print("It is cold outside")
```

```
else:  
    print("It is not too cold today")  
  
# if..elif..else statement  
temperature = 90  
if temperature < 32:  
    print("It is cold outside")  
elif 32 <= temperature <= 80:  
    print("It is a pleasant day today")  
else:  
    print("It is a hot day today")
```

Console Output

```
It is cold outside  
It is not too cold today  
It is a hot day today
```

Loops

Computer programs can execute code very quickly and are ideal for executing the same code in a loop. Python supports multiple types of loops:

- for loop
- while loop

Create a new program called loops.py

```
# for loop  
for i in range(0, 5):  
    print(i)  
  
# while loop  
i = 0  
while i < 5:  
    print(i)  
    i = i + 1
```

Console Output

```
0  
1  
2  
3  
4  
0  
1  
2  
3  
4
```

The for loop will be the most used loop in later project.

Functions

Functions provide a way of "reusing" code. Functions are called from using a function call after a function is defined. They receive data from the program using "arguments" and can return data to the program.

Function syntax:

```
def <function name>(arguments):  
    print(arguments)  
    return 0
```

Create a new program called function.py

```
# Print your name 6 times  
print("Rick")  
print("Rick")  
print("Rick")  
print("Rick")  
print("Rick")  
print("Rick")
```

```
def print_name(name):
    for i in range(0,6):
        print(name)

print_name("Rick")
```

Console Output:

```
Rick
```

__main__ Function

__main__ is a special type in Python called a "magic method". I use this method to define what to do if a program or module is run as the main program. It works well for testing purposes and setting up a large program main file.

Create a new program called test_main.py (note that it does not need to be named main.py)

```
# Print your name 6 times
def print_name(name):
    for i in range(0,6):
        print(name)
```

```
# Program main entry point
if __name__ == "__main__":
    print_name("Rick")
```

Console Output

```
Rick
Rick
Rick
Rick
Rick
Rick
```

This allows the program to run as a main file or as a module. We will discuss modules later in the tutorial.

List Data Structure

The list data structure is a collection of data. It is a very powerful feature in Python because the list can consist of any data type and each element can be a different data type. It is like a dynamically sized array but with any data type.

Define an empty list

```
a_list = []
```

Define a list of strings

```
a_list = ["Hello", "Python", "World"]
```

Define a list of data of different types

```
a_list = [1, 53.999, "Stuff"]
```

Lists have many pre-defined functions or methods available, let's write a program to demonstrate some of them.

Write a new program called `test_list.py`

```
# Define a list
a_list = [10, 25.2, "Hey"]

# Access the list elements
print("a_list[0]: ", a_list[0])
print("a_list[1]: ", a_list[1])
print("a_list[2]: ", a_list[2])

# Iterate over a list
for item in a_list:
    print(item)

# Add a boolean item to the end of the list
a_list.append(True)
print(a_list)

# Print the length of the list
print("List length:", len(a_list))

# Create a list of lists
list_of_lists = [
    [1, 2, 3],
    ["1", "2", "3"],
    [True, False, True]
]
print(list_of_lists)

# Access the second item in the second list
print(list_of_lists[1][1])

# Clear a list
a_list.clear()
print("a_list after clear: ", a_list)
```

Console Output

```
a_list[0]: 10
a_list[1]: 25.2
a_list[2]: Hey
10
25.2
Hey
[10, 25.2, 'Hey', True]
List length: 4
[[1, 2, 3], ['1', '2', '3'], [True, False, True]]
2
a_list after clear: []
```

We will use lists extensively in the projects as you will see.

Classes and Objects

Python supports object-oriented programming (OOP) as well as procedural and functional programming. As we will be using OOP extensively in all projects we need to understand classes, objects, inheritance, abstraction, encapsulation, and polymorphism. The process of writing OOP programs includes:

- Define a class
- Instantiate (create) an object from the class
- Use the object like a custom data type in the program

Let's write a program to demonstrate basic OOP called test_oop.py

```
# Define a class
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.loc = (x, y)

    print("Shape Class initialized")

    def draw(self):
        print("Drawing shape at x,y:", self.loc)
```

```
class Rectangle(Shape):
    def __init__(self, x, y):
        super().__init__(x, y)

        print("Rectangle Class initialized")

    def draw(self):
        print("Drawing rectangle at x,y:", self.loc)

# Instantiate the class object called rect
rect = Rectangle(10,10)

# Use the rect object
rect.draw()
```

Console Output

```
Shape Class initialized
Rectangle Class initialized
Drawing rectangle at x,y: (10, 10)
```

We will discuss OOP in more detail as we dive into the projects.

Multithreading

Most modern desktop computers have multiple core processors which is like having more than one computer in a single microprocessor. Python has a `threading` library which allows you to setup a separate "thread" than the main program thread. One way that multithreading can be used is when we need a continuous operation that if implemented in the main thread would greatly impact the performance of the program.

Lets create a program that toggles the state of a variable between True and False at a known rate or frequency.

First we need to import the `threading` and `time` libraries.

```
import threading  
import time
```

Next we setup a global variable called `state` and set it to False.

```
import threading  
import time  
  
state = False
```

We define a function called `background_calculation()` which sets the timer `sleep()` function to 1 second. When the timer has finished the function toggles the state variable to True or False depending on its current state. Finally, we recursively call the `background_calculation()` function and the process starts over.

```
def background_calculation():  
    # set the time  
    time.sleep(1)  
  
    # Toggle and print state  
    global state  
    state = not state  
    print(state)  
    background_calculation()
```

In the main entry point to the program, we define a thread and set the target to the `background_calculation()` function. Next we start the thread which runs the function in a new thread. The thread runs continuously, in this case.

```
import threading  
import time  
  
state = False  
  
def background_calculation():  
    # set the time  
    time.sleep(1)
```

```
# Toggle and print state
global state
state = not state
print(state)
background_calculation()

def main():
    thread = threading.Thread(target=background_calculation)
    thread.start()

if __name__ == '__main__':
    main()
```

Console Output

```
True
False
True
False
True
False
True
False
True
False
True
```

Modules

In large, advanced programming projects, the number of lines of code can become large and difficult to manage. At this point in the project development, it is a good idea to break the project into modules that are easier to maintain and manage. Modular programming also enables reusable code that can be shared with other projects, programmers, and teams. The Python module and packaging system can be difficult to master so for this basic tutorial we will create a single module file and use "import" to load the module into the main file.

Create a module file called shape.py in the same directory with the other files

```
# Define a class
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.loc = (x, y)

    print("Shape Class initialized")

    def draw(self):
        print("Drawing shape at x,y:", self.loc)

class Rectangle(Shape):
    def __init__(self, x, y):
        super().__init__(x, y)

    print("Rectangle Class initialized")

    def draw(self):
        print("Drawing rectangle at x,y:", self.loc)
```

Now, create another file called test_shape.py

```
from shape import Rectangle

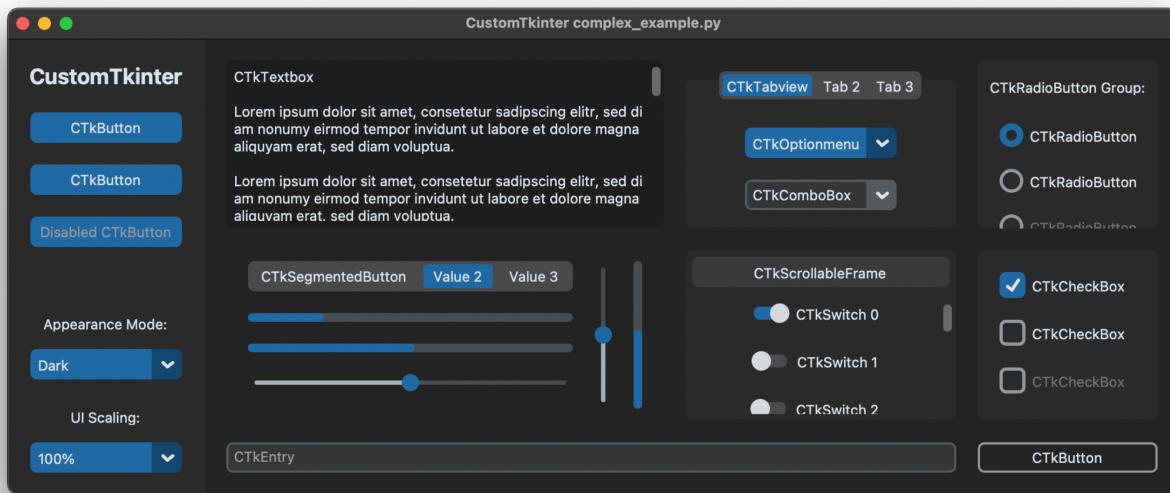
# Instantiate the class object called rect
rect = Rectangle(10,10)

# Use the rect object
rect.draw()
```

Note that the shape classes are defined in a module or library and are "imported" into the main test shape file.

GUI Programming

Console programs such as those created in this tutorial are not very interesting. Most if not all modern programs provide a graphic user interface (GUI) that is user-friendly. One such GUI library is tkinter which ships with the Python installation, however the tkinter graphics are dated. I will use a modern version of tkinter called [CustomTkinter](#) created by Tom Schimanski. As you can see from the image below taken from the documentation, it is much nicer.



As we will be using CustomTkinter for all projects, let's write a simple GUI program using OOP that displays a button that when pressed prints a message to the console.

Create a new program called test_ctk.py

```
import customtkinter as ctk

class App(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("400x150")
        self.title("Test CustomTkinter")

        self.button = ctk.CTkButton(self, text="my button",
command=self.button_callback)
        self.button.pack(padx=20, pady=20)

    def button_callback(self):
        print("button clicked")
```

```
app = App()  
app.mainloop()
```

Before we can run the program, we need to install the CustomTkinter library, open the Terminal tab and enter:

```
py -m pip install customtkinter
```

```
D:\EETools\Python3Tutorial>  
D:\EETools\Python3Tutorial>py -m pip install customtkinter  
Collecting customtkinter  
  Using cached customtkinter-5.2.0-py3-none-any.whl (295 kB)  
Collecting darkdetect  
  Using cached darkdetect-0.8.0-py3-none-any.whl (9.0 kB)  
Installing collected packages: darkdetect, customtkinter  
Successfully installed customtkinter-5.2.0 darkdetect-0.8.0  
  
[notice] A new release of pip available: 22.2.2 -> 23.3  
[notice] To update, run: D:\Pythons\Python39\python.exe -m pip install --  
upgrade pip
```

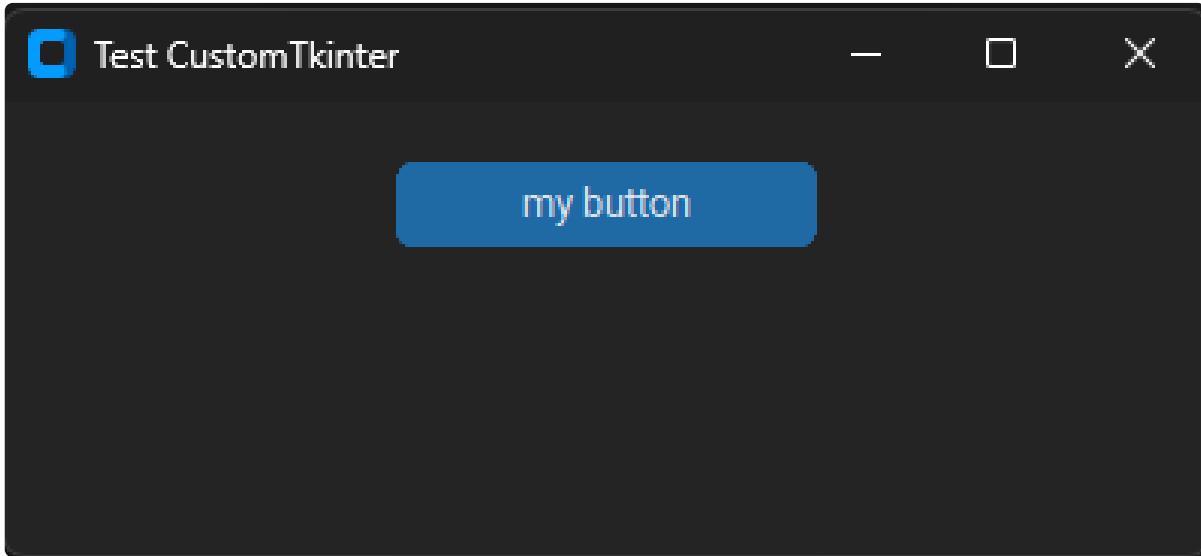
It is a good idea to update pip if a new release is available

```
D:\EETools\Python3Tutorial>python.exe -m pip install --upgrade pip  
Requirement already satisfied: pip in d:\pythons\python311\lib\site-packages  
(23.2.1)  
Collecting pip  
  Obtaining dependency information for pip from  
  https://files.pythonhosted.org/packages/e0/63/b428aaca15fc98c39b07ca7149e24bc1  
  4205ad0f1c80ba2b01835aedde1/pip-23.3-py3-none-any.whl.metadata  
  Downloading pip-23.3-py3-none-any.whl.metadata (3.5 kB)  
Using cached pip-23.3-py3-none-any.whl (2.1 MB)  
Installing collected packages: pip  
Attempting uninstall: pip  
  Found existing installation: pip 23.2.1  
  Uninstalling pip-23.2.1:
```

```
Successfully uninstalled pip-23.2.1
```

```
Successfully installed pip-23.3
```

Now, run the program



Console Output after clicking the button

```
button clicked
```

Summary

This concludes our whirlwind tour of the basic Python tutorial. If you need additional tutorials and resources to learn python, please refer the Appendix on Python References. If you are like me, it is time to start creating useful programs that every Electrical Engineer uses in his daily routine. We will start with the beginner project on creating a scientific calculator in the next chapter.

Chapter 2 - Scientific Calculator Project

This chapter will discuss the creation of a Scientific Calculator using Python and CustomTkinter. A good practice in software design is to define the initial features and specifications of the calculator application before starting development.

Features and Specifications:

- ✓ Numeric keys - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- ✓ Decimal point
- ✓ Sign (+/-)
- ✓ Basic math - addition, subtraction, multiplication, division
- ✓ ln
- ✓ log
- ✓ 10^x
- ✓ square root
- ✓ Parentheses - ()
- ✓ n!
- ✓ x^2
- ✓ $1/x$
- ✓ absolute value
- ✓ exp
- ✓ mod(%)
- ✓ pi
- ✓ e
- ✓ clear
- ✓ backspace
- ✓ Trigonometry
 - ✓ sin, cos, tan
 - asin, acos, atan
 - ✓ hyp
 - ✓ sec, csc, cot
 - ✓ radians, degrees
- ✓ Memory
- ✓ MC

- MR
 - M+
 - M-
 - MS
- Wide screen design
 - Assume 10 rows x 10 columns

Optional Features

- Graphing
 - Programming Language - Python?
 - Large equation library
 - RPN or Algebraic data entry
-

Project Setup

Open PyCharm and create a new project called ScientificCalculator using Python 3.11. From a terminal in PyCharm, install CustomTkinter

```
pip install customtkinter
```

The installation may already be satisfied by the installation during the Python3Tutorial.

Project Directory: D:/EETools/ScientificCalculator

GUI Design

CustomTkinter has 3 layout options: place, pack, and grid. For a calculator, the grid layout is optimal for a calculator interface. As defined in the specification, the initial grid design is 10 rows x 10 columns. The top two rows will be the display area. Rows 3 to 10 are buttons. Some of the buttons may be two columns in width or two rows in height.

Create a new file called scientific_calculator.py. We could create each button individually and assign it to a grid position. A more efficient method is to create and array of buttons

and an array of button text which makes it easy for the programmer to visualize the buttons and assign test.

```
# Initialize a button array (list of lists)
self.buttons = [[None]*10 for _ in range(5)]

# Create buttons
self.button_text = [
    ["sin", "cos", "tan", "hyp", "MC", "M+", "M-", "MS", "bs"], # Row 5
    ["sec", "csc", "cot", "eng", "sci", "clr", "7", "8", "9", "/"], # Row 6
    ["exp", "mod", "pi", "e", "x<>y", "4", "4", "5", "6", "*"], # Row 7
    ["x^2", "1/x", "|x|", "n!", "()", "1", "2", "3", "-"], # Row 8
    ["ln", "log", "10^x", "x^y", "sqrt", "+/-", "0", "=", ".", "+"], # Row 9
]
```

We can display the 5x10 button array using two for loops.

```
# Display button in a 5x10 array
for x in range(5):
    for y in range(10):
        self.buttons[x][y] = ctk.CTkButton(self, text=self.button_text[x][y],
height=60,
                           font=('arial', 16),
                           command=lambda x1=x, y1=y:
self.press(x1, y1))
        if 0 < x < 5 < y < 9:
            self.buttons[x][y].configure(fg_color="#3b3b3b")
        else:
            self.buttons[x][y].configure(fg_color="#7b7b7b")
        self.buttons[x][y].grid(row=x+4, column=y, padx=5, pady=5)
```

Finally, we need a single button callback function that can select the button that was pressed and print the button text, for now.

```
def press(self, x, y):
    print(self.button_text[x][y])
```

Nice! Here is the complete program
scientific_calculator.py

```

import customtkinter as ctk

class App(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("1525x460x10x10")
        self.title("Scientific Calculator")

        calculation = ""

        # Create a label to display results of the calculation
        label = ctk.CTkLabel(self, text="0.000000", padx=10, anchor=ctk.SE,
font=('arial',22), fg_color="transparent",
                           width=1500, height=60)
        label.grid(row=0, column=0, columnspan=10, rowspan=4, padx=10, pady=5)
        label.configure(bg_color="#555500")

        # Initialize a button array (list of lists)
        self.buttons = [[None]*10 for _ in range(5)]

        # Create buttons
        self.button_text = [
            ["sin", "cos", "tan", "hyp", "MC", "MR", "M+", "M-", "MS", "bs"],
# Row 5
            ["sec", "csc", "cot", "eng", "sci", "clr", "7", "8", "9", "/"], #
Row 6
            ["exp", "mod", "pi", "e", "x<>y", "4", "4", "5", "6", "*"], # Row
7
            ["x^2", "1/x", "|x|", "n!", "(", ")", "1", "2", "3", "-"], # Row 8
            ["ln", "log", "10^x", "x^y", "sqrt", "+/-", "0", "=", ".", "+"], #
Row 9
        ]

        # Display button in a 5x10 array
        for x in range(5):
            for y in range(10):
                self.buttons[x][y] = ctk.CTkButton(self,
text=self.button_text[x][y], height=60,
                                             font=('arial',16),
                                             command=lambda x1=x, y1=y:
self.press(x1, y1))
                if 0 < x < 5 < y < 9:
                    self.buttons[x][y].configure(fg_color="#3b3b3b")
                else:
                    self.buttons[x][y].configure(fg_color="#7b7b7b")

```

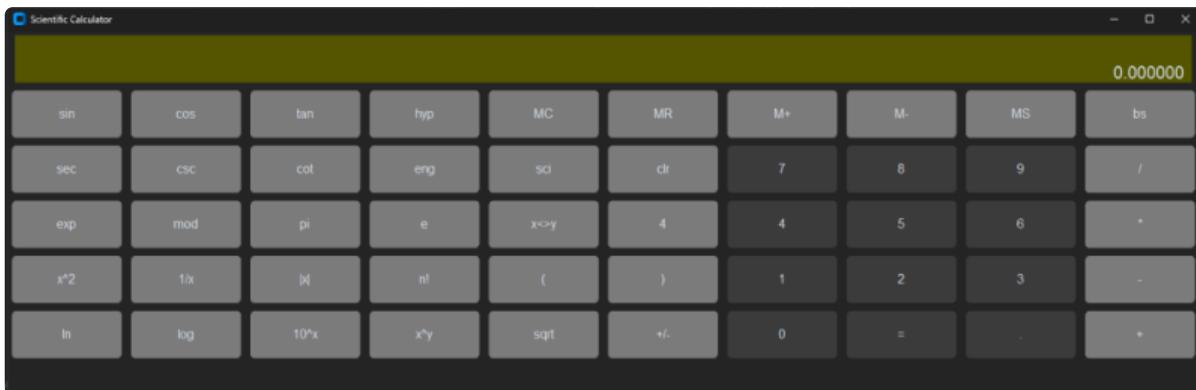
```

        self.buttons[x][y].grid(row=x+4, column=y, padx=5, pady=5)

    def press(self, x, y):
        print(self.button_text[x][y])

if __name__ == "__main__":
    app = App()
    app.mainloop()

```



Here are some additional tweaks:

- ✓ Adjust the window height
- ✓ Find unicode text for divide symbol
- ✓ Find unicode text for backspace key
- ✓ Find unicode text for square root key
- ✓ Find text for PI symbol

After a few tweaks to the button text array

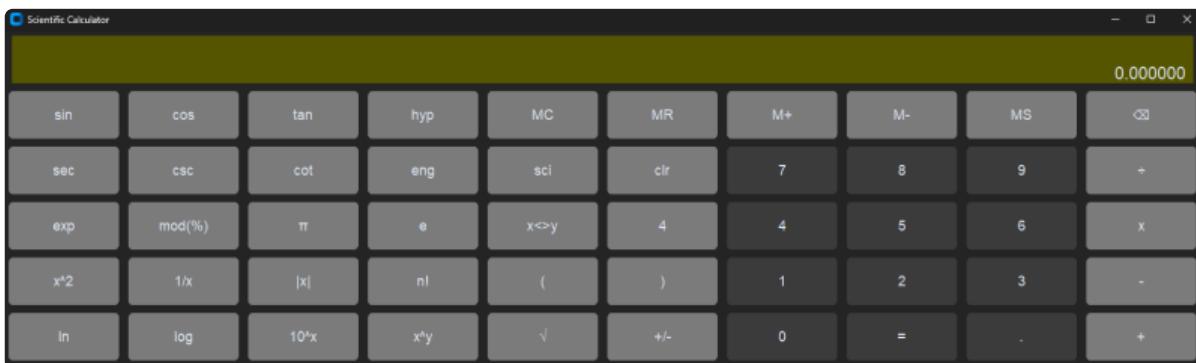
```

# Create buttons
self.button_text = [
    ["sin", "cos", "tan", "hyp", "MC", "MR", "M+", "M-", "MS",
"\u232B"], # Row 5
    ["sec", "csc", "cot", "eng", "sci", "clr", "7", "8", "9",
"\u00f7"], # Row 6
    ["exp", "mod(%)", "\u03c0", "e", "x>>y", "4", "4", "5", "6", "x"],
# Row 7
    ["x^2", "1/x", "|x|", "n!", "(", ")", "1", "2", "3", "-"], # Row 8
    ["ln", "log", "10^x", "x^y", "\u221a", "+/-", "0", "=", ".", "+"],

```

```
# Row 9
```

```
]
```



With the initial user-interface defined, we can proceed to implement the calculator logic.

Calculator Logic

The key to calculating the result of the expression in the label is to process `eval(expression)` if possible. `eval` will evaluate the expression using the Python math library but the expression must use the syntax defined for the [math library](#). Many of the keys have the correct syntax however some have special cases to improve the text on the button or expression that cannot be directly calculated by `eval()`. Memory buttons are also a special case.

The label class has an argument called "textvariable" which can be set to a special string variable called "ctk.StringVar". The result of the `eval()` calculation will be stored in this variable called "self.equation". `eval()` will be called when the user presses the "=" key. A string variable called "self.expression" will contain the expression in the label to be evaluated, i.e. `eval(self.expression)`. Memory value will be stored in "self.memory".

```
self.expression = ""
self.equation = ctk.StringVar()
self.sel = ""
self.memory = 0
```

Here is the update label definition with the textvariable set to self.equation.

```
label = ctk.CTkLabel(self, text="0.00", padx=10, anchor=ctk.SE, font= ('arial',22), textvariable=self.equation,
```

```
fg_color="transparent", width=1500, height=60)
```

Updated button text array

```
self.button_text = [
    ["sin", "cos", "tan", "\u03c0", "MC", "MR", "M+", "M-", "MS",
"\u232b"], # Row 5
    ["asin", "acos", "atan", "radians", "degrees", "CLR", "7", "8",
"9", "\u00f7"], # Row 6
    ["sec", "csc", "cot", "e", "exp", "%", "4", "5", "6", "*"], # Row
7
    ["x^2", "1/x", "|x|", "n!", "()", "1", "2", "3", "-"], # Row 8
    ["log", "log10", "10^x", "2\u03c0", "\u221a", "+/-", "0", "=", ".",
"+"], # Row 9
]
```

The press() function will check for any special cases, memory operations, or simply update the label expression.

```
def press(self, x, y):
    # Check for special cases where eval can't evaluate the button text
    self.sel = self.button_text[x][y] # self.sel is a short alias for
    self.button_text[x][y]
    if self.sel == "CLR" or self.sel == "=" or self.sel == "\u00f7" or self.sel
    == "1/x" or \
        self.sel == "\u232b" or self.sel == "\u03c0" or self.sel == "x^2"
    or self.sel == "|x|" or \
        self.sel == "+/-" or self.sel == "n!" or self.sel == "10^x" or
    self.sel == "\u221a" or \
        self.sel == "2\u03c0":
        self.special_case(x, y)
    elif self.sel == "MC" or self.sel == "MR" or self.sel == "M+" or self.sel
    == "M-" or self.sel == "MS":
        self.memory_operation(x,y)
    else:
        self.expression += self.button_text[x][y]
        self.equation.set(self.expression)
```

The special_case() function detects the key pressed and processes the expression as needed. For example, pi is defined as a unicode character "\u03c0" on the key which

displays the Greek letter for pi. However, math.pi is substituted for the Greek character in the expression.

```
def special_case(self, x, y):
    if self.sel == "CLR": # Clear button selected
        self.expression = ""
        self.equation.set("")
    elif self.sel == "=": # Equal button pressed, evaluate the current
        expression
        total = str(eval(self.expression))
        self.equation.set(total)
        self.expression = ""
    elif self.sel == "\u00f7": # Divide button pressed
        self.expression += "/"
        self.equation.set(self.expression)
    elif self.sel == "1/x": # 1/x button pressed
        self.expression = "1 / " + self.expression
        self.equation.set(self.expression)
    elif self.sel == "\u232b": # backspace button pressed
        self.expression = self.expression.rstrip(self.expression[-1])
        self.equation.set(self.expression)
    elif self.sel == "\u03c0": # pi
        self.expression += "pi"
        self.equation.set(self.expression)
    elif self.sel == "2\u03c0": # 2pi
        self.expression += "2*pi"
        self.equation.set(self.expression)
    elif self.sel == "x^2": # x raised to the power of 2
        self.expression = pow(float(self.expression), 2)
        self.equation.set(str(self.expression))
    elif self.sel == "|x|": # abs(x)
        self.expression = fabs(float(self.expression))
        self.equation.set(str(self.expression))
    elif self.sel == "+/-": # invert sign
        self.expression = -float(self.expression)
        self.equation.set(str(self.expression))
    elif self.sel == "n!": # factorial
        self.expression = factorial(int(self.expression))
        self.equation.set(str(self.expression))
    elif self.sel == "10^x": # 10 to the power of x
        self.expression = pow(10, float(self.expression))
        self.equation.set(str(self.expression))
    elif self.sel == "\u221a": # square root
```

```

        self.expression = sqrt(float(self.expression))
        self.equation.set(str(self.expression))

```

`memory_operation()` function detects which memory key is pressed and processes the operation for that key.

```

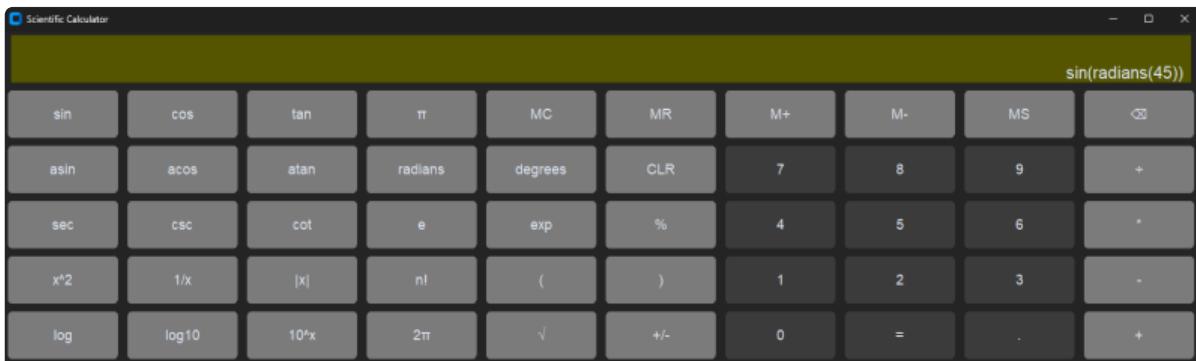
def memory_operation(self, x, y):
    if self.sel == "MC":
        self.memory = 0
    elif self.sel == "MR":
        self.expression += str(self.memory)
        self.equation.set(str(self.expression))
    elif self.sel == "M+":
        self.memory += int(self.expression)
    elif self.sel == "M-":
        self.memory -= int(self.expression)
    elif self.sel == "MS":
        self.memory = int(self.expression)

```

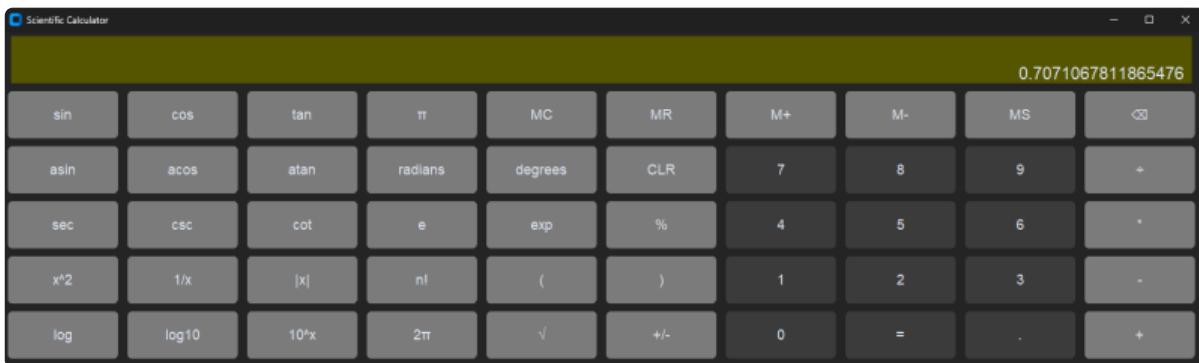
Note that trig functions such as sine and cosine must be entered as follows:

- Press the sin key
- Press the left parenthesis key
- Press the desired value key (value must be in radians)
- Press the right parenthesis key
- Press the = key or add more to the expression

Calculator before hitting the = key.



Calculator after hitting the = key.



Congratulations on completing the Scientific Calculator project. The complete source code is 119 software lines of code (SLOC) and is shown below.

`scientific_calculator.py`

```
import customtkinter as ctk
from math import *

class App(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("1525x425x10x10")
        self.title("Scientific Calculator")

        self.expression = ""
        self.equation = ctk.StringVar()
        self.sel = ""
        self.memory = 0

        # Create a label to display results of the calculation
        label = ctk.CTkLabel(self, text="0.00", padx=10, anchor=ctk.SE, font=('arial',22), textvariable=self.equation,
                             fg_color="transparent", width=1500, height=60)
        label.grid(row=0, column=0, columnspan=10, rowspan=4, padx=10, pady=5)
        label.configure(bg_color="#555500")

        # Initialize a button array (list of lists)
        self.buttons = [[None]*10 for _ in range(5)]

        # Create buttons
        self.button_text = [
            ["sin", "cos", "tan", "\u03c0", "MC", "MR", "M+", "M-", "MS",
             "\u232b"], # Row 5
            ["asin", "acos", "atan", "radians", "degrees", "CLR", "7", "8",
             "9", "\u00f7"], # Row 6
            ["sec", "csc", "cot", "e", "exp", "%", "4", "5", "6", "*"], # Row 7
            ["log", "log10", "10^x", "2\u03c0", "\u221a", "+/-", "0", "=",
             ".", "+"], # Row 8
            [".", "+/-", "C", "CE", "AC", "OFF", "DEG", "RAD", "F1", "F2"] # Row 9
        ]
```

7

```
    ["x^2", "1/x", "|x|", "n!", "()", "1", "2", "3", "-"], # Row 8
    ["log", "log10", "10^x", "2\u03c0", "\u221A", "+/-", "0", "=", ".", "+"], # Row 9
]

# Display button in a 5x10 array
for x in range(5):
    for y in range(10):
        self.buttons[x][y] = ctk.CTkButton(self,
text=self.button_text[x][y], height=60,
font=('arial',18),
command=lambda x1=x, y1=y:
self.press(x1, y1))
        if 0 < x < 5 < y < 9:
            self.buttons[x][y].configure(fg_color="#3b3b3b")
        else:
            self.buttons[x][y].configure(fg_color="#7b7b7b")
        self.buttons[x][y].grid(row=x+4, column=y, padx=5, pady=5)

def press(self, x, y):
    # Check for special cases where eval can't evaluate the button text
    self.sel = self.button_text[x][y] # self.sel is a short alias for
self.button_text[x][y]
    if self.sel == "CLR" or self.sel == "=" or self.sel == "\u00f7" or
self.sel == "1/x" or \
        self.sel == "\u232B" or self.sel == "\u03c0" or self.sel ==
"x^2" or self.sel == "|x|" or \
        self.sel == "+/-" or self.sel == "n!" or self.sel == "10^x" or
self.sel == "\u221A" or \
        self.sel == "2\u03c0":
        self.special_case(x, y)
    elif self.sel == "MC" or self.sel == "MR" or self.sel == "M+" or
self.sel == "M-" or self.sel == "MS":
        self.memory_operation(x,y)
    else:
        self.expression += self.button_text[x][y]
        self.equation.set(self.expression)

def special_case(self, x, y):
    if self.sel == "CLR": # Clear button selected
        self.expression = ""
        self.equation.set("")
    elif self.sel == "=": # Equal button pressed, evaluate the current
expression
        total = str(eval(self.expression))
```

```

        self.equation.set(total)
        self.expression = ""
    elif self.sel == "\u00F7": # Divide button pressed
        self.expression += "/"
        self.equation.set(self.expression)
    elif self.sel == "1/x": # 1/x button pressed
        self.expression = "1 / " + self.expression
        self.equation.set(self.expression)
    elif self.sel == "\u232B": # backspace button pressed
        self.expression = self.expression.rstrip(self.expression[-1])
        self.equation.set(self.expression)
    elif self.sel == "\u03C0": # pi
        self.expression += "pi"
        self.equation.set(self.expression)
    elif self.sel == "2\u03C0": # 2pi
        self.expression += "2*pi"
        self.equation.set(self.expression)
    elif self.sel == "x^2": # x raised to the power of 2
        self.expression = pow(float(self.expression), 2)
        self.equation.set(str(self.expression))
    elif self.sel == "|x|": # abs(x)
        self.expression = fabs(float(self.expression))
        self.equation.set(str(self.expression))
    elif self.sel == "+/-": # invert sign
        self.expression = -float(self.expression)
        self.equation.set(str(self.expression))
    elif self.sel == "n!": # factorial
        self.expression = factorial(int(self.expression))
        self.equation.set(str(self.expression))
    elif self.sel == "10^x": # 10 to the power of x
        self.expression = pow(10, float(self.expression))
        self.equation.set(str(self.expression))
    elif self.sel == "\u221A": # square root
        self.expression = sqrt(float(self.expression))
        self.equation.set(str(self.expression))

def memory_operation(self, x, y):
    if self.sel == "MC":
        self.memory = 0
    elif self.sel == "MR":
        self.expression += str(self.memory)
        self.equation.set(str(self.expression))
    elif self.sel == "M+":
        self.memory += int(self.expression)
    elif self.sel == "M-":
        self.memory -= int(self.expression)

```

```
    elif self.sel == "MS":  
        self.memory = int(self.expression)  
  
if __name__ == "__main__":  
    app = App()  
    app.mainloop()
```

Summary

This tutorial covered the design and development of a scientific calculator program. This is a beginner project because the total lines of code are fairly small. In the next chapters, we will move on to intermediate projects to create a shape editor and a line editor which will form the basis for a diagram editor similar to Microsoft Visio. Electrical Engineering simulators are essentially schematics (diagrams) with analysis capability.

Chapter 3 - Shape Editor Project

With a functional scientific calculator under our belts, so to speak, lets proceed forward and start a new project called Shape Editor. This project has intermediate difficulty primarily because of the size of the project code base. The Shape Editor application is a 2D drawing program written in Python using the tkinter canvas shape library which includes the following shapes:

- Rectangle
- Oval
- Polygon
- Line
- Arc
- Text
- Image

The name Tkinter, which ships with every Python installation, comes from TK interface, a Python binding to the [Tk GUI toolkit](#) and was written by Guido van Rossum (the father of Python) among others. CustomTkinter uses the Tkinter canvas which has its own shape management capability based on a shape ID assigned to each shape. We will create our own shape management library using object-oriented programming (OOP). It is educational to learn to make your own class libraries that you manage and maintain. This application will allow us to explore the 2D shape library listed above except for lines. The Line Editor project in the next chapter will explore line drawing using the mouse and how to manage the interaction between shapes and lines.

Project Design, Features, and Specification

Required features:

- ✓ Modern GUI look and feel
 - ✓ Canvas
 - ✓ Left frame with shape menu
 - ✓ Top frame with shape controls
- ✓ Canvas appearance
 - ✓ Background grid

- Snap to Grid
- 2D Shape Classes
 - Rectangle
 - Oval
 - Polygon
- Shape manipulation
 - Move
 - Rotate
 - Resize
- Shape appearance
 - Fill color
 - Border color
 - Border width
- Modularized source code using Python modules and packages
- Object-oriented programming (OOP)

Shape Creation and Drawing Algorithm

We need a graphical drawing algorithm or code design that will create shape objects and display them on the computer screen. We need to "persist" the objects when the screen is resized or moved.

List of steps to create and draw a shape on the canvas:

- Shape selection from shape menu
- Draw the shape using the mouse which creates a shape object
- Save the shape in a shape list
- Iterate over the shape list and call each shape's draw() method to display it on the canvas

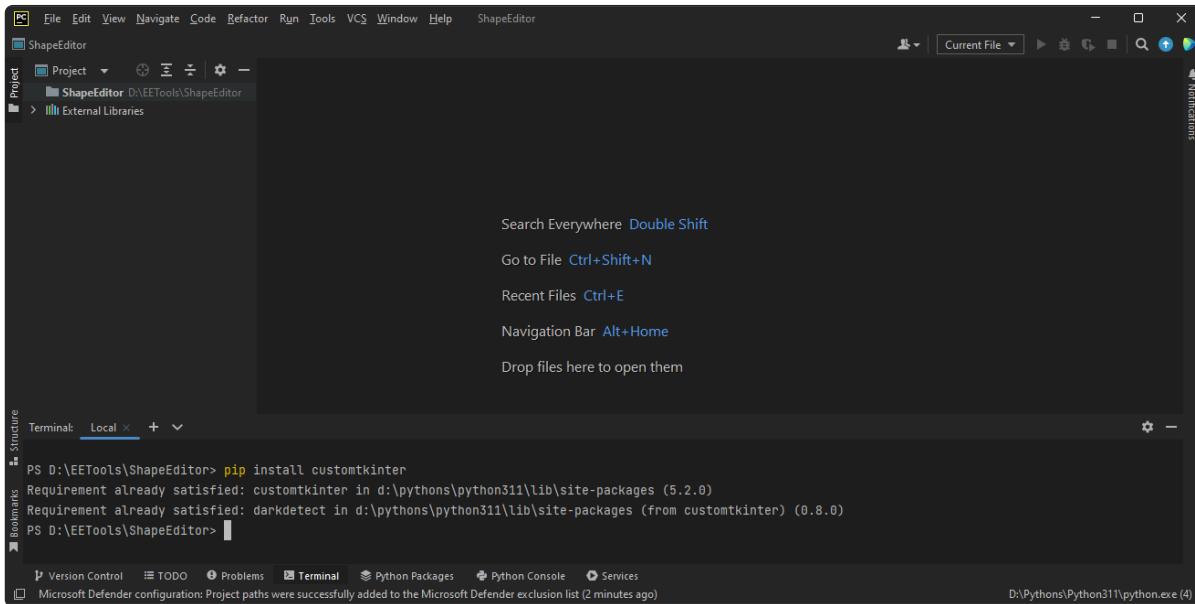
Project Setup

Enough talk, let's create a new project.

- Open PyCharm and create a new project called ShapeEditor using Python 3.11.5 interpreter in the "venv"
- From a terminal in PyCharm, install CustomTkinter

- pip install customtkinter
 - The installation may already be satisfied by the installation during the previous projects.

My Project Directory: D:/EETools/ShapeEditor



GUI Design and Layout

Goal: Create the initial GUI layout in a CustomTkinter window and draw a red rectangle.

Create the main application file in ShapeEditor called `shape_editor.py`.

During the GUI design phase of development I like to create a "mock-up" of the GUI layout using Python comments. This keeps the proposed layout design documented in the code file rather than some external drawing.

```
# Main Window Frame Design
#
# ##### Top Frame #####
# #
# # Left   Canvas
# # Frame #
```

```
# #           #
# ##### ##### ##### ##### ##### ##### ##### ##### ##### ##### ##### ##### #####
```

First import the CustomTkinter library with an alias of ctk. We use aliases instead of the actual name because it is much shorter.

```
import customtkinter as ctk
```

Create the application class which inherits the [ctk window class](#). Essentially we are creating a custom ctk window class for this program. It inherits all the functionality of the ctk window class. We can now initialize the size, position, and title of the application window. The initial size of the window is 1200x800 pixels. The initial position of the window is x, y = 10, 10. The geometry is set by passing a string in the format "`<Width>x<Height>x<x>x<y>`". The window title is "Shape Editor".

```
class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100")  # w, h, x, y
        self.title("Shape Editor")
```

Finally, create the program main entry point, create a application object, and run the main loop.

```
if __name__ == "__main__":
    """Instantiate the Shape Editor application and run the main loop"""
    app = ShapeEditorApp()
    app.mainloop()
```

shape_editor.py - complete program

```
# Main Window Frame Design
#
# ##### ##### ##### ##### ##### ##### ##### ##### ##### ##### ##### ##### #####
```

```
# ######
# #      #
# #      #
# # Left   #          Canvas      #
# # Frame  #          #
# #      #          #
# ######



import customtkinter as ctk


class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        # Add widgets to the app here


    if __name__ == "__main__":
        """Instantiate the Shape Editor application and run the main loop"""
        app = ShapeEditorApp()
        app.mainloop()
```

Run the program and a new blank window is shown. The application icon is the default ctk icon at this point. We can change this in the future.



Draw a Red Rectangle

Canvas

To draw a red rectangle, we need a 2D drawing surface called a [canvas](#). CTk uses the tkinter canvas and its documentation. A canvas is a container for graphical objects, objects that can be displayed on the computer screen.

We create a canvas in the application class initialization code as follows:

```
class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        # Add widgets to the app here
        self.canvas = ctk.CTkCanvas()
```

Widgets are not shown on the computer screen until we add the widget to a layout manager. Tkinter provides 3 layout managers: place, pack, and grid. Place positions the widget at absolute coordinates or position on the screen. We have already seen the grid layout manager in the Scientific Calculator project. We will use the ["pack" layout manager](#) for this application which works well for application GUI layout.

Pack or Packer Layout Manager

The pack layout manager specifies a "relative" position for the widget in a parent container such as top, left, right, bottom, etc. It also has options to fill and expand the widget within its parent. Other options include internal and external padding.

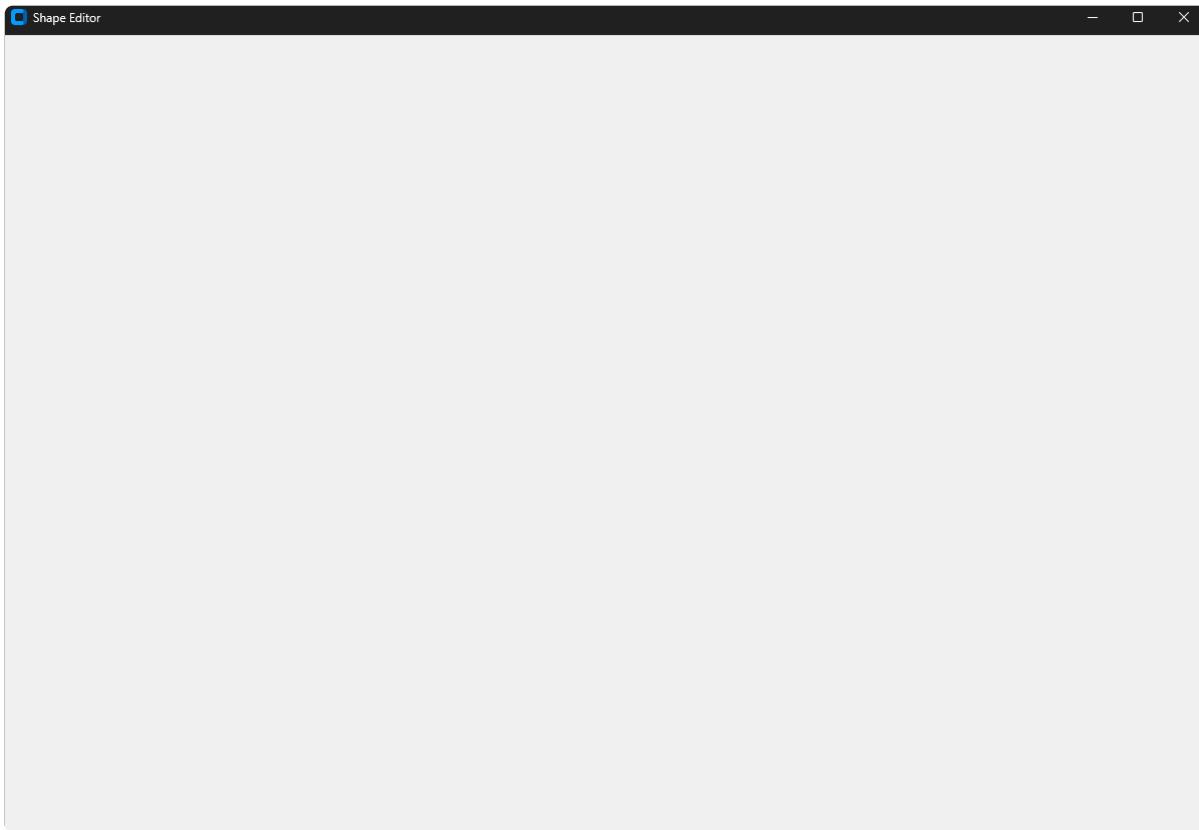
Add a pack statement to display the canvas.

```
# Add widgets to the app here
self.canvas = ctk.CTkCanvas()
self.canvas.pack(fill="both", expand=True, padx=2, pady=2)
```

pack options:

- `fill="both"` tells the interpreter to fill the parent container in both the x and y directions
- `expand=True` causes the canvas to automatically expand the canvas size if the parent container size is changed
- `padx=2` adds 2 pixels of "padding" to the right and left sides (x directions) of the canvas
- `pady=2` adds 2 pixels of "padding" to the top and bottom sides (y directions) of the canvas

Run the program, the white area on the screen is the drawing canvas which is now displayed.



Drawing a Shape

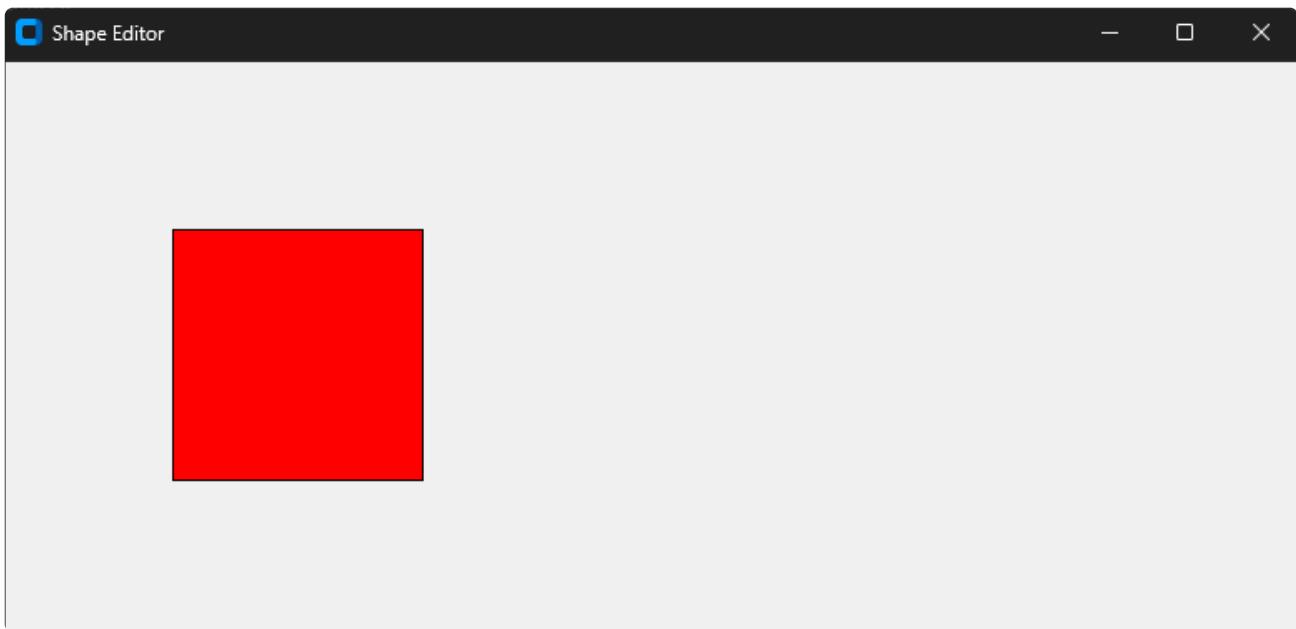
The canvas class has a library for creation of shapes:

- `canvas.create_rectangle(x1, y1, x2, y2, option=value)`
- `canvas.create_oval(x1, y1, x2, y2, option=value)`
- `canvas.create_polygon(points_list, option=value)`
- `canvas.create_line(x1, y1, x2, y2, option=value)`
- `canvas.create_image(x, y, option=value)`
- `canvas.create_text(x, y, option=value)`

For this OOP application, the canvas creation function is "wrapped" in a custom class for each shape. For now, let's add a statement to draw a red rectangle on the canvas.

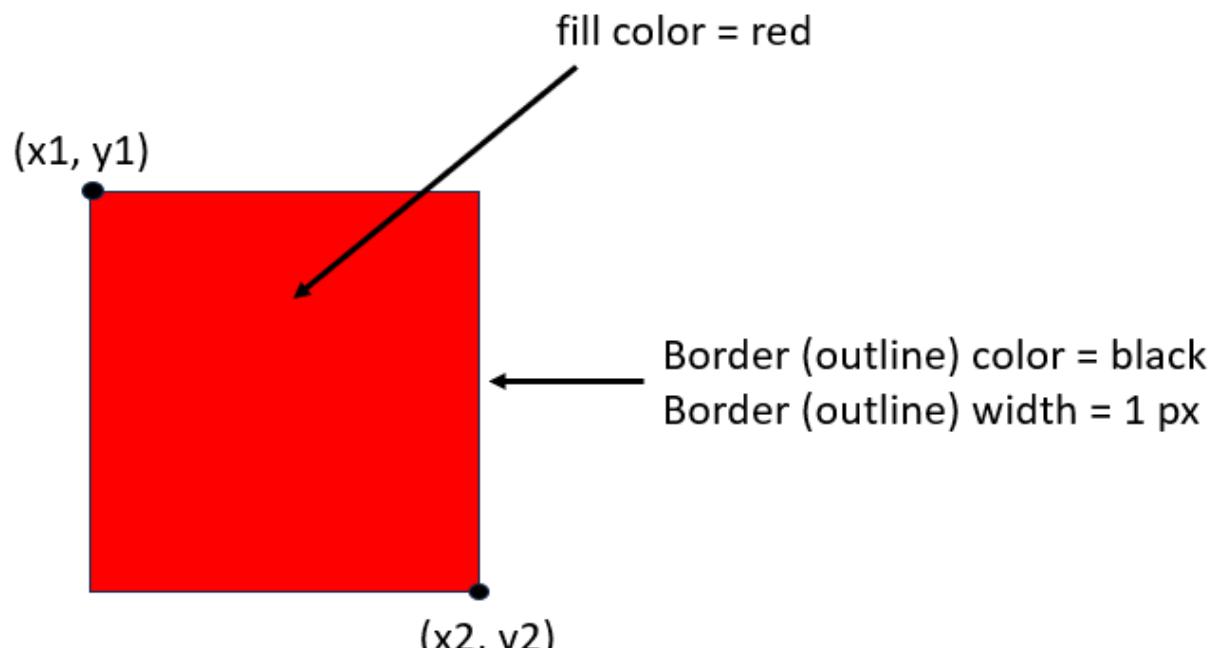
```
# Add widgets to the app here
self.canvas = ctk.CTkCanvas()
self.canvas.pack(fill="both", expand=True, padx=2, pady=2)

# Draw a red rectangle
self.canvas.create_rectangle(100, 100, 250, 250, fill="red")
```



Nice! A red rectangle is displayed. The diagram below shows the anatomy of a rectangle shape.

Rectangle Shape



$$\text{Width} = x_2 - x_1$$

$$\text{Height} = y_2 - y_1$$

The rectangle is drawn from the coordinates representing the upper-right corner (x_1, y_1) to the coordinates representing the lower-left corner (x_2, y_2). Other geometrical parameters can be calculated from the "bounding box" coordinates as follows:

- Width = $x_2 - x_1$
- Height = $y_2 - y_1$

Note that the x coordinates increase in value from the left to right side of the screen. The y coordinates increase in value from the top to the bottom of the screen.

The appearance of the rectangle is defined by optional values.

```
canvas.create_rectangle(x1, y1, x2, y2, fill=fill_color, outline=border_color,  
width=border_width)
```

Three options provide control over the shape appearance:

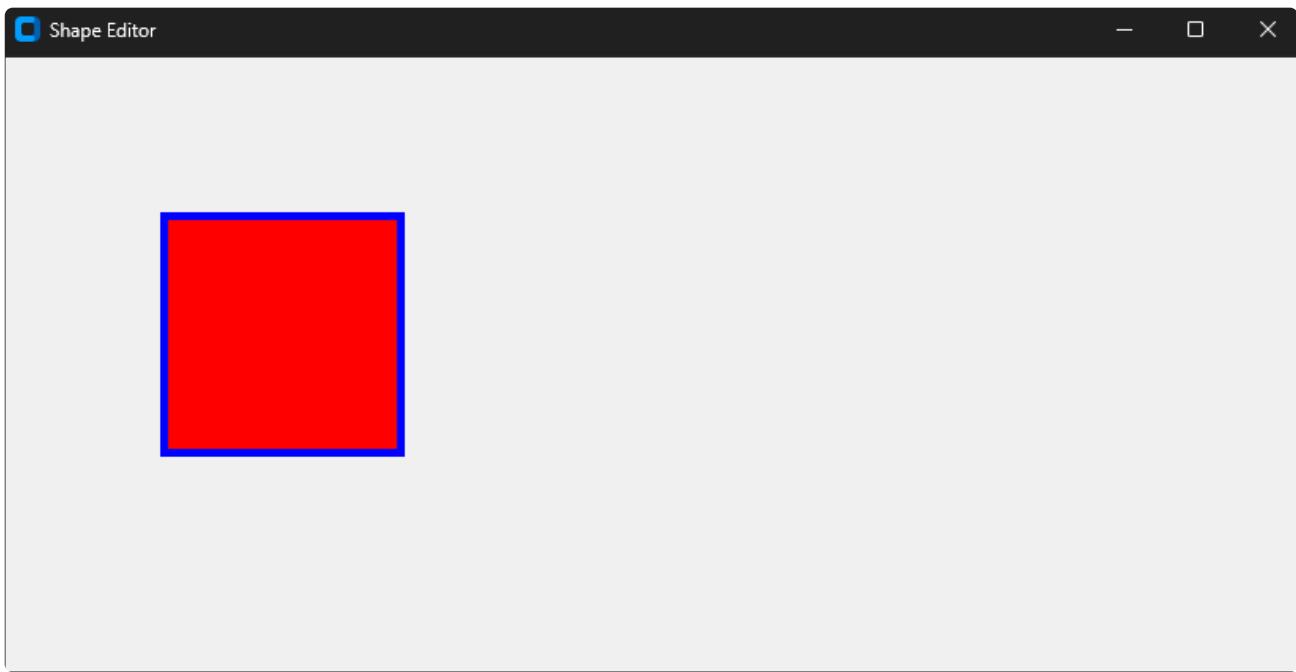
- `fill=fill_color`, example: `fill="red"`
- `outline=border_color`, example: `outline="black"`
- `width=border_width`, example: `width=5`

The "fill" defines the internal color of the shape. There are predefined colors such as "red", "green", and "blue". Colors can be defined as RGB hex values of the form "#RRGGBB" such as "#FF0000" for red, "#00FF00" for green, and "#0000FF" for blue. To set the fill to a transparent color, simply set `fill=None`. Hex values for any colors can be found at <https://www.color-hex.com/>.

Although it is difficult to see in the diagrams above, the rectangle has a black border by default with a border width of 1 px.

Let's change the border (outline) color and border width in our program.

```
# Draw a red rectangle  
self.canvas.create_rectangle(100, 100, 250, 250, fill="red",  
outline="blue", width=5)
```



Now that we know how to control the shape appearance we can add controls to the Shape Editor application to change the values of these options.

Frame

The Frame widget is a container for other widgets and is the primary way of adding widgets to the GUI. Referring to our GUI mock-up, we will add two Frames to our application. The top frame will contain the shape appearance controls to set fill colors, border colors, and border width. The left frame will contain buttons to create the various shapes we want to explore in this project.

The [CTk Frame widget](#) has a required option called `master`. `Master` is the the parent where the frame will be drawn which in our case is the application window. Optional arguments include `width`, `height`, `border width`, `foreground color (fg_color)`, and `border color`. Note that a Frame will adjust its size to fit the widgets added to it.

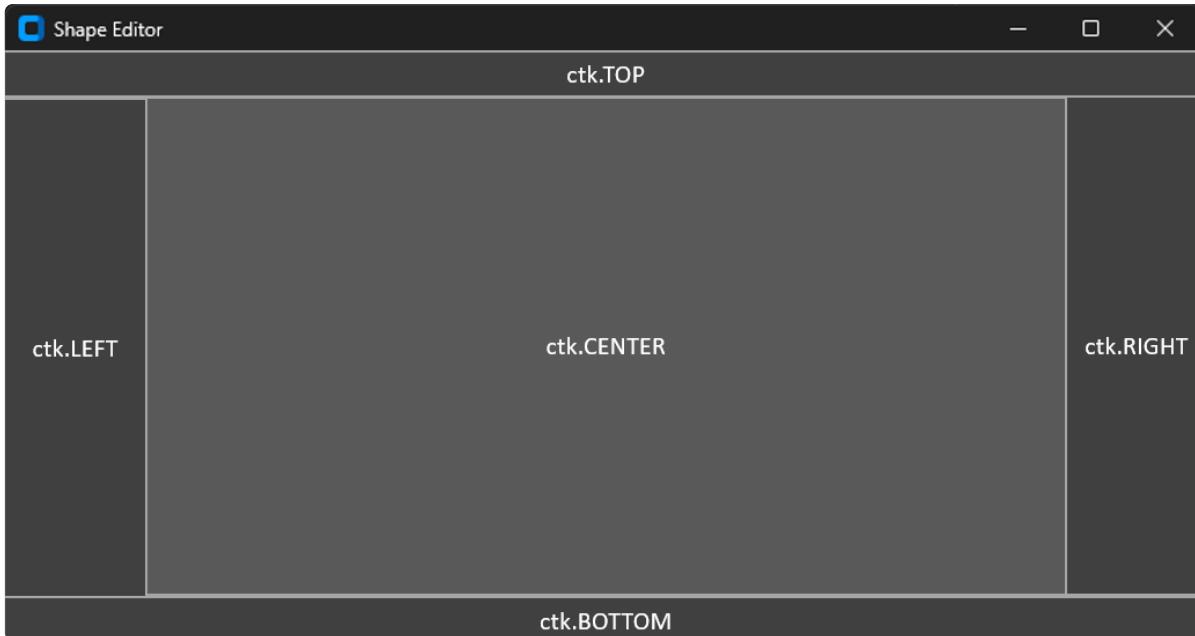
Top Frame

Add a frame called `top_frame` to the application.

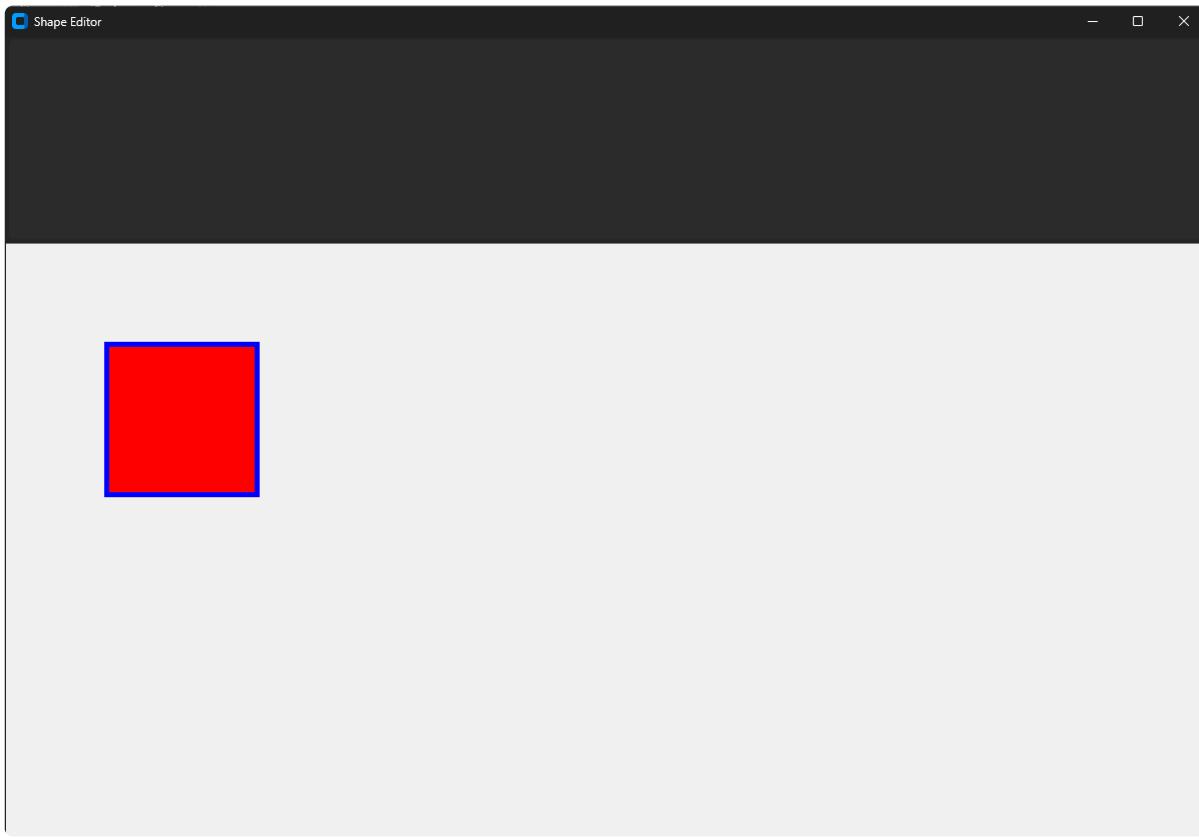
```
# Add widgets to the app here
self.canvas = ctk.CTkCanvas()
self.top_frame = ctk.CTkFrame(master=self)
```

```
self.top_frame.pack(side=ctk.TOP, fill="both", padx=5, pady=2)
self.canvas.pack(fill="both", expand=True, padx=2, pady=2)
```

The option `side=ctk.TOP` tells the packer to place the Frame at the top of the window. CTk includes a set of constants for frame placement: `ctk.TOP`, `ctk.BOTTOM`, `ctk.RIGHT`, `ctk.LEFT` and `ctk.CENTER`. The following diagram shows the placement areas of the window.



Run the program to see the frame displayed at the top of the window.



We do not use the option to expand the frame because we want the frame to adjust to the size of the widgets added to it.

Shape Appearance Controls

The shape appearance controls will be used to set the fill color, border color, and border width of any selected shape. Let's start by creating "buttons" that change the appearance of the rectangle shape. Give the rectangle a name so we can reference it from the button functions. Note that this name will store an integer value called an ID.

```
# Draw a red rectangle
self.rect_id = self.canvas.create_rectangle(100, 100, 250, 250,
                                              fill="red", outline="blue",
                                              width=5)
```

[CTk Button](#) works like most buttons that you have used in any GUI. The one key element of a button is that it has an option to call a function. We will use the button function to set the appearance values of the shape using the canvas configure method.

The signature for the button creation is:

```
button_name = ctk.CTkButton(parent, text="Button Text", command=button_function)
```

We will create a button that when clicked calls a function to set the rectangle fill color to green.

```
# Draw a red rectangle
self.rect_id = self.canvas.create_rectangle(100, 100, 250, 250,
                                             fill="red", outline="blue", width=5)

# Add top frame widgets here
self.fill_button = ctk.CTkButton(self.top_frame, text="Fill",
                                  command=self.set_fill_color)
self.fill_button.pack(side=ctk.LEFT, padx=3, pady=3)

def set_fill_color(self):
    self.canvas.itemconfig(self.rect_id, fill="green")
```

Shape Editor after the button is clicked. Note that the top frame has adjusted its height to the height of the button.



Add two more buttons to change the border color and the border width. This code will change the border color to yellow and the border width to 10 px.

```
# Add top frame widgets here
self.fill_color_button = ctk.CTkButton(self.top_frame,
```

```

        text="Fill Color",
        command=self.set_fill_color)
self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

self.border_color_button = ctk.CTkButton(self.top_frame,
                                         text="Border Color",
                                         command=self.set_border_color)
self.border_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

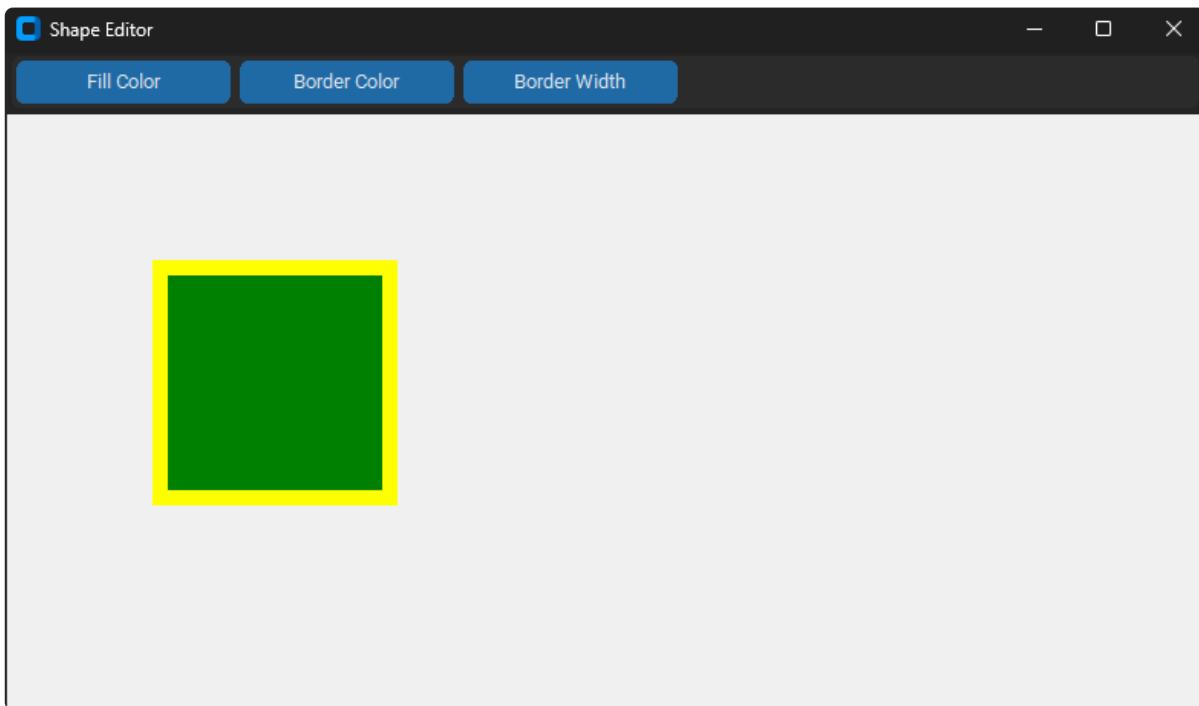
self.border_width_button = ctk.CTkButton(self.top_frame,
                                         text="Border Width",
                                         command=self.set_border_width)
self.border_width_button.pack(side=ctk.LEFT, padx=3, pady=3)

def set_fill_color(self):
    self.canvas.itemconfig(self.rect_id, fill="green")

def set_border_color(self):
    self.canvas.itemconfig(self.rect_id, outline="yellow")

def set_border_width(self):
    self.canvas.itemconfig(self.rect_id, width=10)

```



How can we change this code to allow us to set colors to ANY color and the width to a value in a range from 1 to 10? This process of changing the code is called "refactoring". We will refactor the code many times to improve the code structure or add new features.

You probably have used a "color picker" in other applications to select a color for something. It would be a nice feature to have a color picker for fill and border color selection. Fortunately, there is a nice library called [CTkColorPicker](#) based on the CustomTkinter library. Open a terminal in PyCharm and install it:

```
pip install CTkColorPicker
```

Note that this will also install the popular "pillow" library for image manipulation.

Using the example under "How to use?" on the CTkColorPicker GitHub website, change the fill and border color button code to call a function that displays the color picker and sets the rectangle colors to the user selected colors.

Import the color picker library.

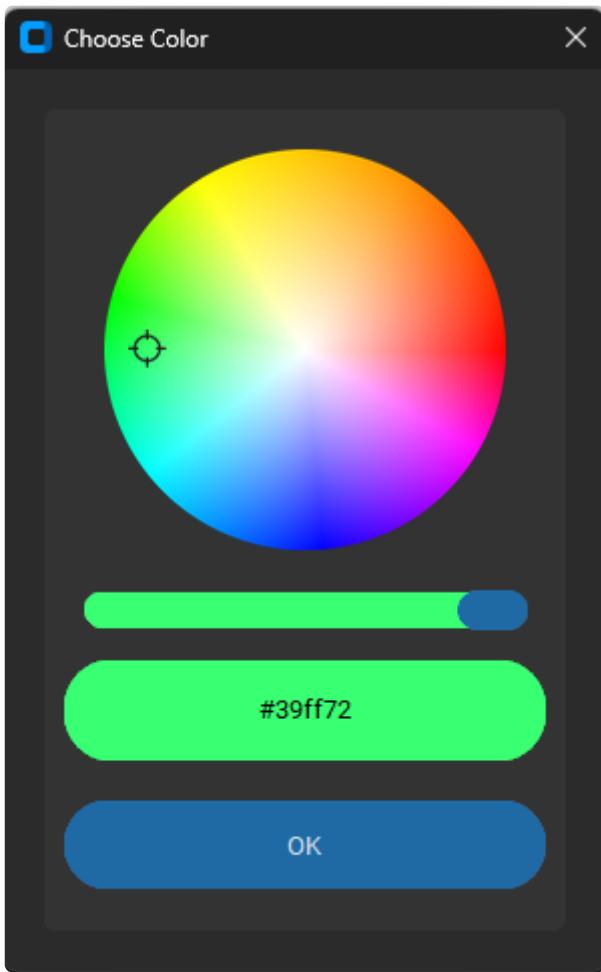
```
import customtkinter as ctk
from CTkColorPicker import *
```

Add two lines of code to the color button functions that use the `AskColor()` function to open the color picker and `pick_color.get()` to store the user color. Then we set the fill color or the border color to the user selected color.

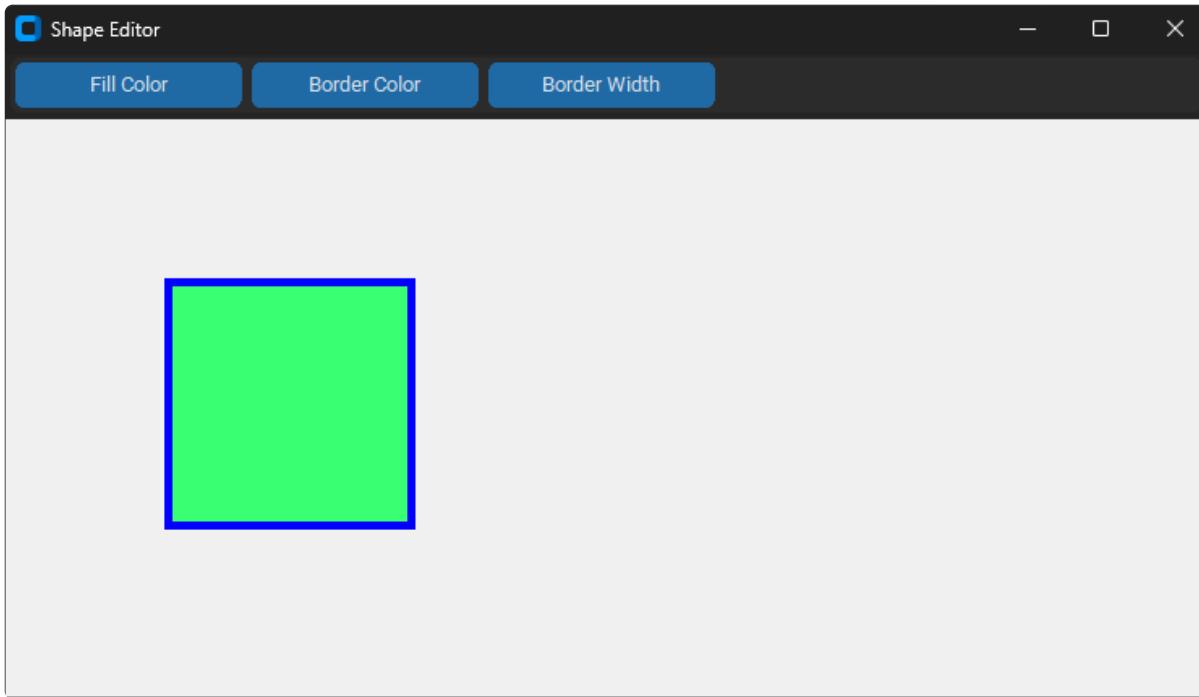
```
def set_fill_color(self):
    pick_color = AskColor() # open the color picker
    color = pick_color.get() # get the color string
    self.canvas.itemconfig(self.rect_id, fill=color)

def set_border_color(self):
    pick_color = AskColor() # open the color picker
    color = pick_color.get() # get the color string
    self.canvas.itemconfig(self.rect_id, outline=color)
```

Color picker - Move the circular target to set the color and move the slider to set the brightness.



Rectangle after clicking the OK button in the color picker.



That is a lot of functionality for just a few lines of code. This is a good example of how a 3rd party library can greatly enhance your program and save you some programming time.

We could use a [CTk Combo Box](#) or a CTk Option Menu(<https://customtkinter.tomschimansky.com/documentation/widgets/optionmenu>) to provide a menu of border width sizes. Lets use the Option Menu for this application. Change the border width button to an option menu and modify the border width button function based on the example code without the variable as follows:

```
# self.border_width_button = ctk.CTkButton(self.top_frame,
#                                         text="Border Width",
#                                         command=self.set_border_width)
# self.border_width_button.pack(side=ctk.LEFT, padx=3, pady=3)

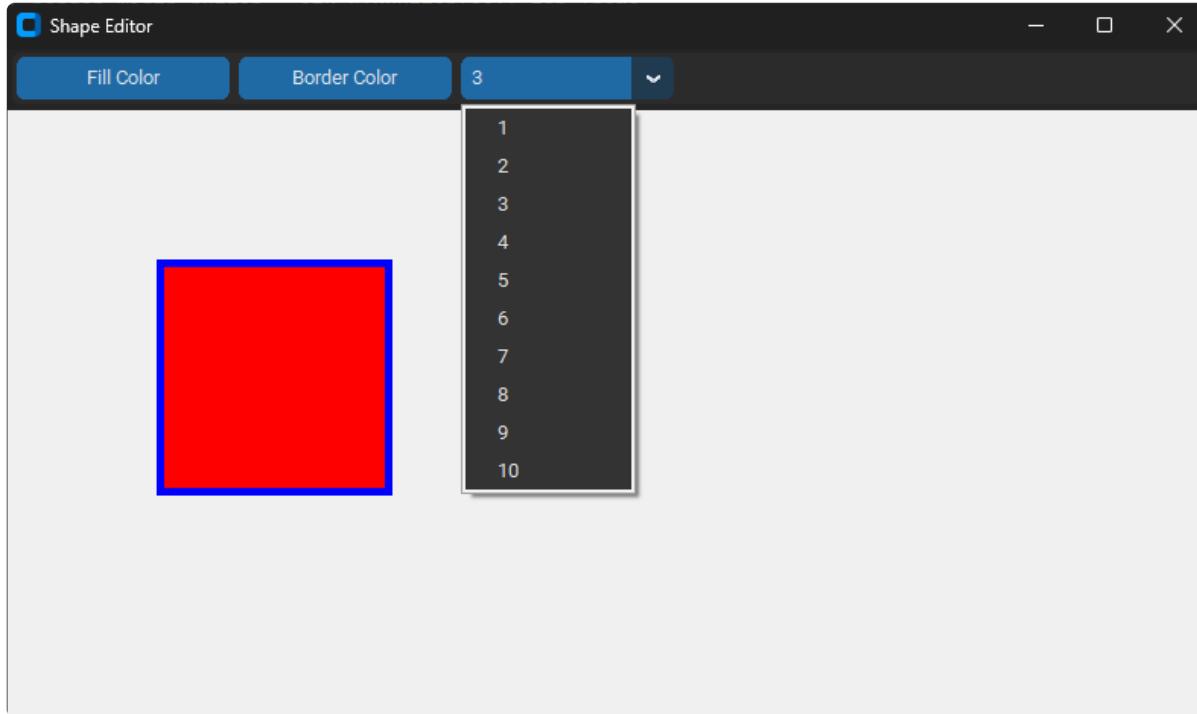
border_width_optionmenu = ctk.CTkOptionMenu(self.top_frame,
                                             values=[ "1", "2", "3", "4", "5", "6", "7", "8", "9",
"10" ],
                                             command=self.set_border_width)
border_width_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
border_width_optionmenu.set("3")
```

Note that I "commented out" the old code. This is a useful practice during refactoring until the new code is functioning properly as you might want to use some of the old code (like the pack options) or if the new code doesn't work you can delete it and uncomment the old code.

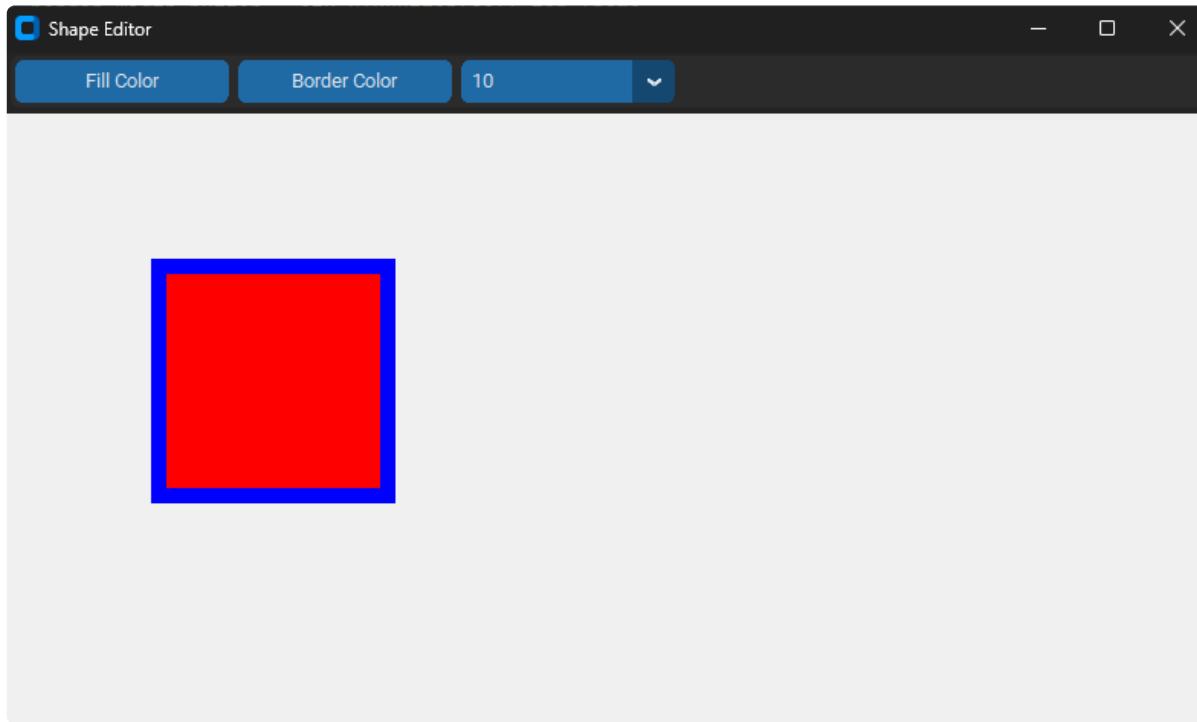
Now, add a choice argument to the `set_border_width()` function and set the width to the choice.

```
def set_border_width(self, choice):
    self.canvas.itemconfig(self.rect_id, width=choice)
```

Main window with the border width option menu selected.



Rectangle with the border width set to 10 px.



Take some time to experiment with setting the colors and width of the rectangle to get familiar with the new controls. We will continue to improve the shape appearance controls but first it is time to start looking at shape classes and shape creation.

Classes and Object-Oriented Programming (OOP)

Suppose that we want to create more than one rectangle and other shapes like a circle or triangle, how do we change our program to handle this? We can use custom classes to create our own library of shapes with features and capabilities required by the application.

Lets take a deep dive into class creation, instantiation, and object usage. Note that we have been using a class to define the Shape Editor Application. Lets create a rectangle class above the code for the Shape Editor Application class.

```
class Rectangle:  
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,  
                 border_width):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.fill_color = fill_color  
        self.border_color = border_color  
        self.border_width = border_width
```

Using standard python practice, we define the class with the name Rectangle, which is capitalized. The class does not inherit from a base class, for now. In the class initializer, we pass the canvas where the rectangle will be drawn, bounding box coordinates, and option for fill color, border color, and border width.

```
def draw(self):  
    self.id = self.canvas.create_rectangle(100, 100, 250, 250,  
                                           fill=self.fill_color,  
                                           outline=self.border_color,  
                                           width=self.border_width)
```

Next, we add a class method called draw() that will draw the shape on the canvas using the canvas create_rectangle command. We capture the shape id in the self.id variable so we can use canvas commands to configure the shape.

Here is the complete class definition:

```
class Rectangle:  
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,
```

```

        border_width):
    self.canvas = canvas
    self.x1 = x1
    self.y1 = y1
    self.x2 = x2
    self.y2 = y2
    self.fill_color = fill_color
    self.border_color = border_color
    self.border_width = border_width

    self.id = None

def draw(self):
    self.id = self.canvas.create_rectangle(100, 100, 250, 250,
                                           fill=self.fill_color,
                                           outline=self.border_color,
                                           width=self.border_width)

```

Next we instantiate an object from our Rectangle class in the main program. Note that we use the new option name that we defined in our custom class to set the appearance of the rectangle. Now we don't have to remember that outline = border and that width = border width.

```

# Draw a red rectangle
self.rect_obj = Rectangle(self.canvas, 100, 100, 250, 250,
                           fill_color="red",
                           border_color="blue",
                           border_width=5)
self.rect_obj.draw()

```

If we run the program now, we get an error because we don't have a `self.rect_id` variable needed to configure the shape. Remember that we created a `self.id` variable in the shape which can be used for this purpose. Modify the shape appearance functions as follows:

```

def set_fill_color(self):
    pick_color = AskColor() # open the color picker
    color = pick_color.get() # get the color string
    self.canvas.itemconfig(self.rect_obj.id, fill=color)

def set_border_color(self):

```

```
pick_color = AskColor() # open the color picker
color = pick_color.get() # get the color string
self.canvas.itemconfig(self.rect_obj.id, outline=color)

def set_border_width(self, choice):
    self.canvas.itemconfig(self.rect_obj.id, width=choice)
```

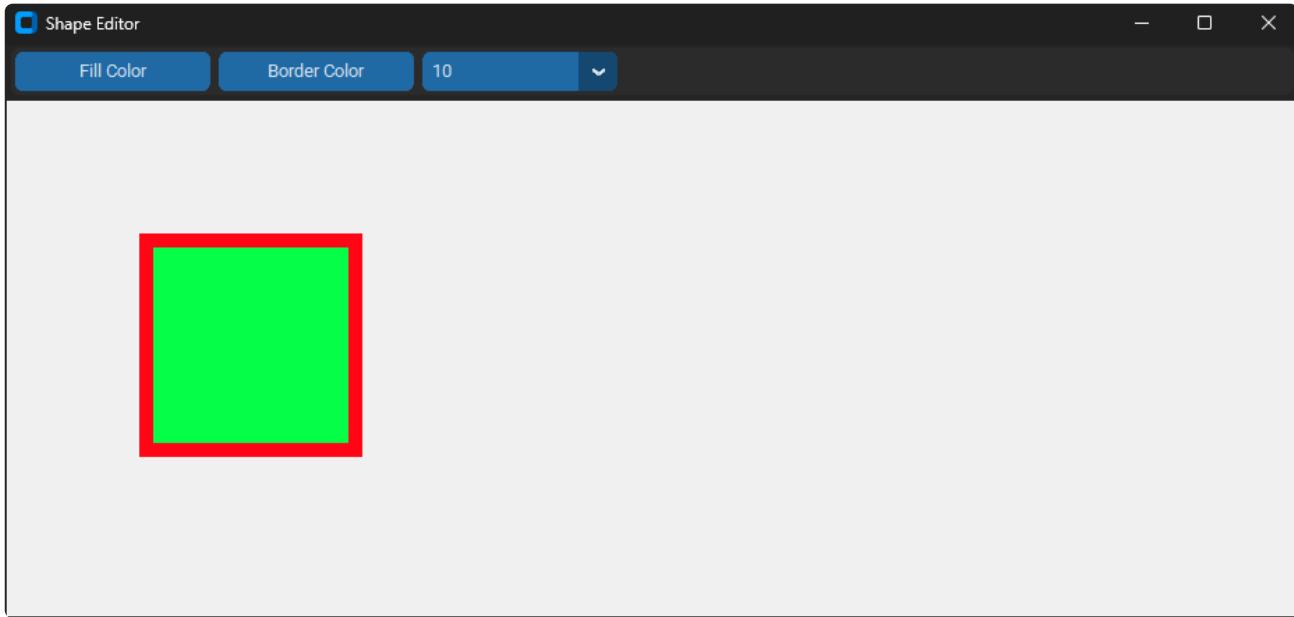
If we run the program now, it runs with no errors and has the same functionality as before. However, we don't need to use the `canvas.itemconfig` method because we now have a custom shape class with the ability to set the shape colors and border width. How can we refactor the shape appearance functions to set colors and width directly?

```
def set_fill_color(self):
    pick_color = AskColor() # open the color picker
    color = pick_color.get() # get the color string
    self.rect_obj.fill_color = color
    self.rect_obj.draw()
    # self.canvas.itemconfig(self.rect_obj.id, fill=color)

def set_border_color(self):
    pick_color = AskColor() # open the color picker
    color = pick_color.get() # get the color string
    self.rect_obj.border_color = color
    self.rect_obj.draw()
    # self.canvas.itemconfig(self.rect_obj.id, outline=color)

def set_border_width(self, choice):
    self.rect_obj.border_width = choice
    self.rect_obj.draw()
    # self.canvas.itemconfig(self.rect_obj.id, width=choice)
```

Testing the application again, we find that it runs as expected. We could delete the `self.id` variable but we will keep it around in case we want to use a canvas method that needs it in the future. Hint, we may need it to select the shape.



Let's add Oval and Triangle classes to our application. The Oval class is pretty easy if we copy the Rectangle class and change the class name and the `canvas.create` methods. The Triangle class requires that we create a polygon with a list of points that define the polygon.

Oval class

Did you catch the error with the Rectangle class? I left hard-coded numbers in the rectangle creation code. To move the rectangle to any position on the canvas, we need to use the coordinates that were passed into the class initialization.

Updated Rectangle draw() method.

Triangle class

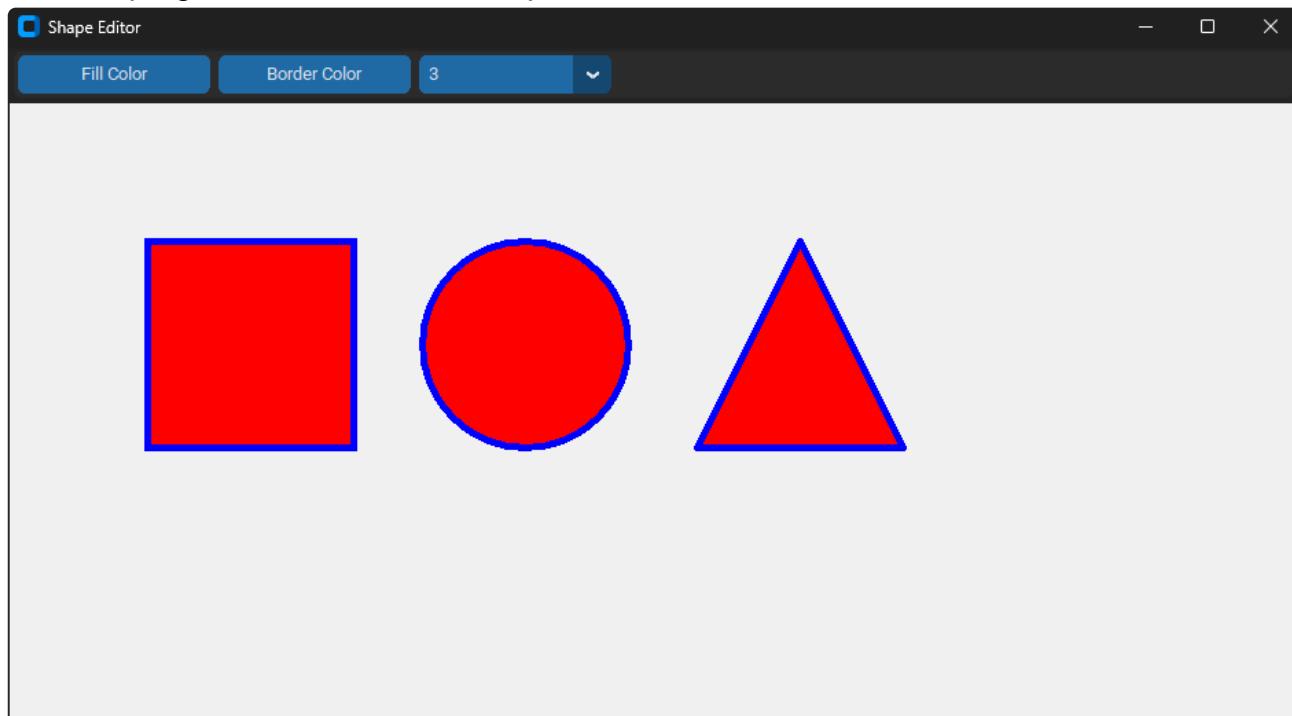
Draw all three shapes on the canvas by creating shape objects for each one, then call the draw() method for that object. We need to space the shapes by adjusting the x1 and x2 coordinates.

```
# Draw a rectangle, oval, and triangle
self.rect_obj = Rectangle(self.canvas, 100, 100, 250, 250,
                           fill_color="red",
                           border_color="blue",
                           border_width=5)
self.rect_obj.draw()

self.oval_obj = Oval(self.canvas, 300, 100, 450, 250,
                      fill_color="red",
                      border_color="blue",
                      border_width=5)
self.oval_obj.draw()

self.tri_obj = Triangle(self.canvas, 500, 100, 650, 250,
                        fill_color="red",
                        border_color="blue",
                        border_width=5)
self.tri_obj.draw()
```

Run the program to see all three shapes



Our shape classes have a lot of duplicated code in each class. We could use "inheritance" to create a "base class" for our shapes and move common code to the base class.

Create a new class called Shape and modify the other classes:

```

class Triangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,
                 border_width):
        super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color,
                        border_width)
        self.points = []

    def draw(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        x1, y1 = (self.x1, self.y1 + h)
        x2, y2 = (self.x1 + w, self.y1 + h)
        x3, y3 = (self.x1 + w / 2, self.y1)
        points = [x1, y1, x2, y2, x3, y3]
        self.id = self.canvas.create_polygon(points,
                                              fill=self.fill_color,
                                              outline=self.border_color,
                                              width=self.border_width)

```

Notice that the classes that inherit from the Shape class have the following statement;

```
super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color, border_width)
```

This passes the initialization arguments to the base class (Shape). Triangle keeps the `self.points` variable because it is unique to the Triangle class. This is a good example of how inheritance helps the programming eliminate duplicate code. Programmers call this the "DRY" principle, where DRY stands for Don't Repeat Yourself.

Wait! We have a problem. We can only change the appearance of the rectangle. We need a way to select a shape and change the appearance for the selected shape.

Shape Selection

Lets write a custom mouse class that will select a shape object when the user presses the left mouse button while hovering over the shape. Although canvas has a method for detecting items with the mouse (`canvas.find_overlapping`), I prefer to write a mouse function that will select a shape object but we will need to write our own "hit" test function.

First lets create a shape list that contains a list of the current shapes on the canvas. Having the ability to iterate over the shapes on the canvas will be a key capability as we add more functionality.

```
class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""

    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        self.shape_list = []

    # Add widgets to the app here
```

Next, add the shapes to the shape_list using the append() function.

```
# Draw a rectangle, oval, and triangle
self.rect_obj = Rectangle(self.canvas, 100, 100, 250, 250,
                           fill_color="red",
                           border_color="blue",
                           border_width=5)

self.rect_obj.draw()
self.shape_list.append(self.rect_obj)

self.oval_obj = Oval(self.canvas, 300, 100, 450, 250,
                     fill_color="red",
                     border_color="blue",
                     border_width=5)

self.oval_obj.draw()
self.shape_list.append(self.oval_obj)

self.tri_obj = Triangle(self.canvas, 500, 100, 650, 250,
                        fill_color="red",
                        border_color="blue",
                        border_width=5)

self.tri_obj.draw()
self.shape_list.append(self.tri_obj)
```

Test the shape list by adding a `print(self.shape_list)` command. Verify that the list is printed in the console. After verification, you can comment out or delete the print statement.

```
[<__main__.Rectangle object at 0x000001E43B780950>, <__main__.Oval object at
0x000001E43B852B50>, <__main__.Triangle object at 0x000001E43BE2F6D0>]
```

What are the canvas mouse events and how do we get access? We need to create a canvas mouse binding to an event handler of the form:

```
self.canvas.bind('<Button-1>', handler)
```

The mouse buttons can be accessed with the following names:

Left mouse button: "<Button-1>"

Middle mouse button: "<Button-2>"

Right mouse button: "<Button-3>"

Lets write some mouse test code under the border width option menu to setup bindings for the 3 mouse buttons.

```
border_width_optionmenu.set("3")

# Canvas mouse bindings
self.canvas.bind('<Button-1>', self.left_mouse_down)
self.canvas.bind('<Button-2>', self.middle_mouse_down)
self.canvas.bind('<Button-3>', self.right_mouse_down)

def left_mouse_down(self, event):
    print("Left click at ", event.x, event.y)

def middle_mouse_down(self, event):
    print("Middle click at ", event.x, event.y)

def right_mouse_down(self, event):
    print("Right click at ", event.x, event.y)
```

Run the program and click on the red rectangle with all 3 mouse buttons. Verify that the print messages appear in the console.

```
Left click at 161 160
Middle click at 161 160
Right click at 161 160
```

Note that not only did the program detect the 3 mouse clicks, it also printed the mouse x, y coordinates. With those coordinates, we can check to see if the mouse is over one on the

shapes on the canvas. To improve the modularity of the program, move the mouse bindings and event handlers to a custom mouse class.

Mouse class

```
class Mouse:  
    def __init__(self, canvas, shape_list):  
        self.canvas = canvas  
        self.shape_list = shape_list  
  
        self.selected_obj = None  
  
        # Canvas mouse bindings  
        self.canvas.bind('<Button-1>', self.left_mouse_down)  
        self.canvas.bind('<Button-2>', self.middle_mouse_down)  
        self.canvas.bind('<Button-3>', self.right_mouse_down)  
  
    def left_mouse_down(self, event):  
        print("Left click at ", event.x, event.y)  
  
    def middle_mouse_down(self, event):  
        print("Middle click at ", event.x, event.y)  
  
    def right_mouse_down(self, event):  
        print("Right click at ", event.x, event.y)
```

The mouse class is initialized with the canvas and shape list and defines a variable called `self.selected_obj` which will contain the selected object. Next, canvas mouse bindings are defined with their associated event handler.

Next instantiate the mouse class in the application after creating the border width option menu.

```
border_width_optionmenu.set("3")  
  
self.mouse = Mouse(self.canvas, self.shape_list)  
  
def set_fill_color(self):
```

If you run the program, it will function as before and print the mouse event messages in the console.

We can now use the event x, y coordinates for shape selection. The select_shape function is a "hit test" function will check to see if the current mouse coordinates are within the boundary or bounding box of a shape on the canvas.

```
def select_shape(self, x, y):
    for s in self.shape_list:
        if (
            s.x <= x <= s.x + s.width
            and s.y <= y <= s.y + s.height
        ):
            self.canvas.selected_obj = s
```

Now modify the left mouse event handler to print a message if a shape is selected.

```
def left_mouse_down(self, event):
    x, y = event.x, event.y
    self.select_shape(x, y)
    if self.selected_obj:
        print("Object found at: ", self.selected_obj, event.x, event.y)
```

Run the program, click on each shape, the following will printed to the console.

```
Object found at: <__main__.Oval object at 0x0000018B1BC8AC50> 388 173
Object found at: <__main__.Triangle object at 0x0000018B1C7E1350> 572 198
Object found at: <__main__.Rectangle object at 0x0000018B1BC76C10> 196 170
```

Next we need to setup the shape appearance controls to configure the selected shape.

```
def set_fill_color(self):
    if self.mouse.selected_obj:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.mouse.selected_obj.fill_color = color
        self.draw_shapes()

def set_border_color(self):
    if self.mouse.selected_obj:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
```

```

        self.mouse.selected_obj.border_color = color
        self.draw_shapes()

def set_border_width(self, choice):
    if self.mouse.selected_obj:
        self.mouse.selected_obj.border_width = choice
        self.draw_shapes()

```

Finally, add a selection box around the shape when it is selected. Modify each shape's draw() method to add a selection box.

Here is the code for the Rectangle class. Add the same to the Oval and Triangle classes.

```

def draw(self):
    self.id = self.canvas.create_rectangle(self.x1, self.y1,
                                            self.x2, self.y2,
                                            fill=self.fill_color,
                                            outline=self.border_color,
                                            width=self.border_width)

    if self.is_selected:
        points = [self.x1-5, self.y1-5, self.x2+5, self.y2+5]
        self.canvas.create_rectangle(points, fill=None, outline="red", width=2)

```

Unselect all other objects so that only one object is selected at a time.

```

def left_mouse_down(self, event):
    if self.selected_obj:
        self.selected_obj.is_selected = False
        self.selected_obj = None
        self.draw_shapes()
    else:
        x, y = event.x, event.y
        self.select_shape(x, y)
        if self.selected_obj:
            print("Object found at: ", self.selected_obj, event.x, event.y)
            self.draw_shapes()

```

```

def select_shape(self, x, y):
    for s in self.shape_list:
        w = s.x2 - s.x1

```

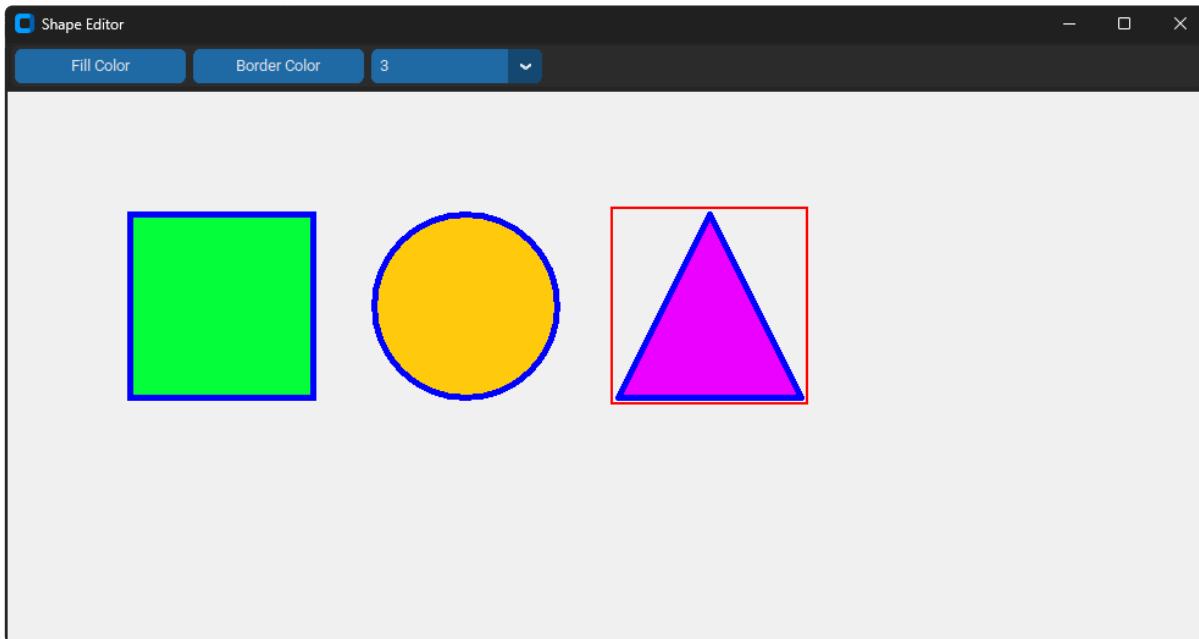
```

        h = s.y2 - s.y1
        if (
            s.x1 <= x <= s.x2 + w
            and s.y1 <= y <= s.y2 + h
        ):
            self.unselect_all_objects()
            self.selected_obj = s
            self.selected_obj.is_selected = True

    def draw_shapes(self):
        self.canvas.delete('all')
        for s in self.shape_list:
            s.draw()

    def unselect_all_objects(self):
        for s in self.shape_list:
            s.is_selected = False
        self.draw_shapes()

```



Cool! We can select any shape and change its appearance. Currently, the diagram shapes are hard-coded in the application. Time to tackle the feature that allows the user to create any shape on the canvas from a menu of shapes.

Left Frame

The shape selection menu will be created in a frame on the left side of the GUI. This is the last GUI element that we need based on the GUI mock-up.

Add a left frame to the Shape Editor App

```
class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        self.shape_list = []

        # Add widgets to the app here
        self.canvas = ctk.CTkCanvas()
        self.top_frame = ctk.CTkFrame(master=self)
        self.left_frame = ctk.CTkFrame(master=self)

        self.top_frame.pack(side=ctk.TOP, fill="both", padx=5, pady=2)
        self.left_frame.pack(side=ctk.LEFT, fill="both", padx=5, pady=2)
        self.canvas.pack(fill="both", expand=True, padx=2, pady=2)
```

Next add three buttons to the frame for the Rectangle, Oval, and Triangle and add the button handles to create the shapes. Delete the old code that created the three shapes automatically.

```
# Add left frame widgets here
rect_button = ctk.CTkButton(self.left_frame,
                            text="Rectangle",
                            command=self.create_rectangle)
rect_button.pack(side=ctk.TOP, padx=5, pady=5)

oval_button = ctk.CTkButton(self.left_frame,
                            text="Oval",
                            command=self.create_oval)
oval_button.pack(side=ctk.TOP, padx=5, pady=5)

tri_button = ctk.CTkButton(self.left_frame,
                           text="Triangle",
                           command=self.create_triangle)
tri_button.pack(side=ctk.TOP, padx=5, pady=5)
```

Note that for a vertical frame the buttons should be placed at the TOP side of the frame.

```

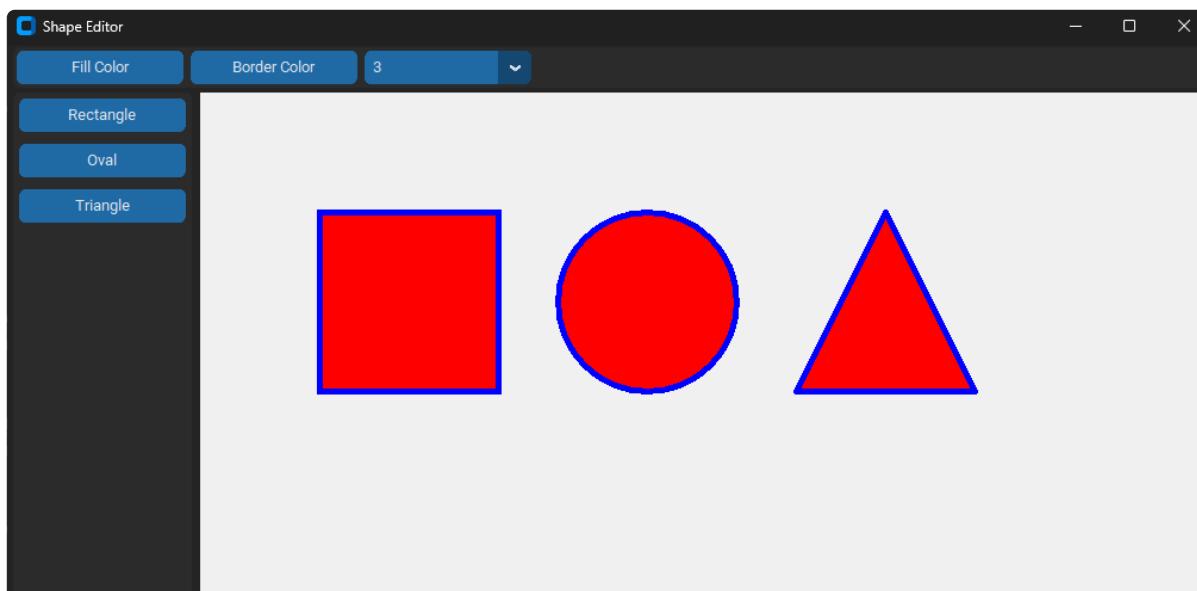
def create_rectangle(self):
    rect_obj = Rectangle(self.canvas, 100, 100, 250, 250,
                         fill_color="red",
                         border_color="blue",
                         border_width=5)
    self.shape_list.append(rect_obj)
    self.mouse.draw_shapes()

def create_oval(self):
    oval_obj = Oval(self.canvas, 300, 100, 450, 250,
                    fill_color="red",
                    border_color="blue",
                    border_width=5)
    self.shape_list.append(oval_obj)
    self.mouse.draw_shapes()

def create_triangle(self):
    tri_obj = Triangle(self.canvas, 500, 100, 650, 250,
                       fill_color="red",
                       border_color="blue",
                       border_width=5)
    self.shape_list.append(tri_obj)
    self.mouse.draw_shapes()

```

The button handlers create a shape object, add it to the shape list, and call the mouse draw_shapes() function which redraws all shapes in the shape list.



Here is the current complete source code for shape_editor.py

```
# Main Window Frame Design
#
# ##### Top Frame #####
# #
# # Left Canvas #
# # Frame #
# #
# ##### #####
#
import customtkinter as ctk
from CTkColorPicker import *


class Shape:
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,
                 border_width):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = fill_color
        self.border_color = border_color
        self.border_width = border_width

        self.id = None
        self.is_selected = False


class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,
                 border_width):
        super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color,
                        border_width)

    def draw(self):
        self.id = self.canvas.create_rectangle(self.x1, self.y1,
                                              self.x2, self.y2,
                                              fill=self.fill_color,
                                              outline=self.border_color,
                                              width=self.border_width)

        if self.is_selected:
```

```

        points = [self.x1-5, self.y1-5, self.x2+5, self.y2+5]
        self.canvas.create_rectangle(points, fill=None, outline="red",
width=2)

class Oval(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,
                border_width):
        super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color,
                        border_width)

    def draw(self):
        self.id = self.canvas.create_oval(self.x1, self.y1, self.x2, self.y2,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

        if self.is_selected:
            points = [self.x1-5, self.y1-5, self.x2+5, self.y2+5]
            self.canvas.create_rectangle(points, fill=None, outline="red",
width=2)

class Triangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,
                 border_width):
        super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color,
                        border_width)
        self.points = []

    def draw(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        x1, y1 = (self.x1, self.y1 + h)
        x2, y2 = (self.x1 + w, self.y1 + h)
        x3, y3 = (self.x1 + w / 2, self.y1)
        points = [x1, y1, x2, y2, x3, y3]
        self.id = self.canvas.create_polygon(points,
                                             fill=self.fill_color,
                                             outline=self.border_color,
                                             width=self.border_width)

        if self.is_selected:
            points = [self.x1-5, self.y1-5, self.x2+5, self.y2+5]
            self.canvas.create_rectangle(points, fill=None, outline="red",
width=2)

```

```
width=2)

class Mouse:
    def __init__(self, canvas, shape_list):
        self.canvas = canvas
        self.shape_list = shape_list

        self.selected_obj = None

    # Canvas mouse bindings
    self.canvas.bind('<Button-1>', self.left_mouse_down)
    self.canvas.bind('<Button-2>', self.middle_mouse_down)
    self.canvas.bind('<Button-3>', self.right_mouse_down)

    def left_mouse_down(self, event):
        if self.selected_obj:
            self.selected_obj.is_selected = False
            self.selected_obj = None
            self.draw_shapes()
        else:
            x, y = event.x, event.y
            self.select_shape(x, y)
            if self.selected_obj:
                # print("Object found at: ", self.selected_obj, event.x,
                event.y)
                self.draw_shapes()

    def middle_mouse_down(self, event):
        print("Middle click at ", event.x, event.y)

    def right_mouse_down(self, event):
        print("Right click at ", event.x, event.y)

    def select_shape(self, x, y):
        for s in self.shape_list:
            w = s.x2 - s.x1
            h = s.y2 - s.y1
            if (
                s.x1 <= x <= s.x2 + w
                and s.y1 <= y <= s.y2 + h
            ):
                self.unselect_all_objects()
                self.selected_obj = s
                self.selected_obj.is_selected = True
```

```

def draw_shapes(self):
    self.canvas.delete('all')
    for s in self.shape_list:
        s.draw()

def unselect_all_objects(self):
    for s in self.shape_list:
        s.is_selected = False
    self.draw_shapes()

class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        self.shape_list = []

        # Add widgets to the app here
        self.canvas = ctk.CTkCanvas()
        self.top_frame = ctk.CTkFrame(master=self)
        self.left_frame = ctk.CTkFrame(master=self)

        self.top_frame.pack(side=ctk.TOP, fill="both", padx=5, pady=2)
        self.left_frame.pack(side=ctk.LEFT, fill="both", padx=5, pady=2)
        self.canvas.pack(fill="both", expand=True, padx=2, pady=2)

        # Add top frame widgets here
        self.fill_color_button = ctk.CTkButton(self.top_frame,
                                              text="Fill Color",
                                              command=self.set_fill_color)
        self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

        self.border_color_button = ctk.CTkButton(self.top_frame,
                                              text="Border Color",
                                              command=self.set_border_color)
        self.border_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

        border_width_optionmenu = ctk.CTkOptionMenu(self.top_frame,
                                                    values=["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"],
                                                    command=self.set_border_width)
        border_width_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
        border_width_optionmenu.set("3")

```

```

# Add left frame widgets here
rect_button = ctk.CTkButton(self.left_frame,
                            text="Rectangle",
                            command=self.create_rectangle)
rect_button.pack(side=ctk.TOP, padx=5, pady=5)

oval_button = ctk.CTkButton(self.left_frame,
                            text="Oval",
                            command=self.create_oval)
oval_button.pack(side=ctk.TOP, padx=5, pady=5)

tri_button = ctk.CTkButton(self.left_frame,
                            text="Triangle",
                            command=self.create_triangle)
tri_button.pack(side=ctk.TOP, padx=5, pady=5)

self.mouse = Mouse(self.canvas, self.shape_list)

def create_rectangle(self):
    rect_obj = Rectangle(self.canvas, 100, 100, 250, 250,
                         fill_color="red",
                         border_color="blue",
                         border_width=5)
    self.shape_list.append(rect_obj)
    self.mouse.draw_shapes()

def create_oval(self):
    oval_obj = Oval(self.canvas, 300, 100, 450, 250,
                    fill_color="red",
                    border_color="blue",
                    border_width=5)
    self.shape_list.append(oval_obj)
    self.mouse.draw_shapes()

def create_triangle(self):
    tri_obj = Triangle(self.canvas, 500, 100, 650, 250,
                       fill_color="red",
                       border_color="blue",
                       border_width=5)
    self.shape_list.append(tri_obj)
    self.mouse.draw_shapes()

def set_fill_color(self):
    if self.mouse.selected_obj:
        pick_color = AskColor() # open the color picker

```

```

        color = pick_color.get() # get the color string
        self.mouse.selected_obj.fill_color = color
        self.draw_shapes()

    def set_border_color(self):
        if self.mouse.selected_obj:
            pick_color = AskColor() # open the color picker
            color = pick_color.get() # get the color string
            self.mouse.selected_obj.border_color = color
            self.draw_shapes()

    def set_border_width(self, choice):
        if self.mouse.selected_obj:
            self.mouse.selected_obj.border_width = choice
            self.draw_shapes()

    def draw_shapes(self):
        self.canvas.delete('all')
        for s in self.shape_list:
            s.draw()

if __name__ == "__main__":
    """Instantiate the Shape Editor application and run the main loop"""
    app = ShapeEditorApp()
    app.mainloop()

```

The source code is still manageable at only 251 lines of code. As you will soon see, the lines of code will increase and the mouse class will have the more lines of code than any of the other classes in the application.

Custom Canvas Class

This is a good point in the development to take a step back and look at the code structure. Is there something out of place for example the mouse class currently draws all the shapes. Lets create a custom Canvas class and put the shape list and the shape drawing functions in it. Also, the mouse class will become a member of the canvas class. This will help decrease the size of mouse class which will help make room for new mouse features: draw shapes, move shapes, resize shapes and rotate shapes with snap-to-grid capability. This is the most difficult part of the Shape Editor development but we will take it one step at a time.

Canvas class

```
class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()

        self.shape_list = []
        self.mouse = Mouse(self)

    def draw_shapes(self):
        self.delete('all')
        for s in self.shape_list:
            s.draw()
```

That was easy except all the variables that pointed to the shape list and the draw shape functions need to be updated such that they are accessed from the canvas class. Also, the mouse is now a member of the canvas class so we don't need to instantiate in the application class - just instantiate the canvas and the mouse comes along with it.

Refactored Application Class

```
class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""

    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        self.shape_list = []

        # Add widgets to the app here
        self.canvas = Canvas()
        self.top_frame = ctk.CTkFrame(master=self)
        self.left_frame = ctk.CTkFrame(master=self)

        self.top_frame.pack(side=ctk.TOP, fill="both", padx=5, pady=2)
        self.left_frame.pack(side=ctk.LEFT, fill="both", padx=5, pady=2)
        self.canvas.pack(fill="both", expand=True, padx=2, pady=2)

        # Add top frame widgets here
        self.fill_color_button = ctk.CTkButton(self.top_frame,
```

```
        text="Fill Color",
        command=self.set_fill_color)
self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

self.border_color_button = ctk.CTkButton(self.top_frame,
                                         text="Border Color",
                                         command=self.set_border_color)
self.border_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

border_width_optionmenu = ctk.CTkOptionMenu(self.top_frame,
                                             values=["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"],
                                             command=self.set_border_width)
border_width_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
border_width_optionmenu.set("3")

# Add left frame widgets here
rect_button = ctk.CTkButton(self.left_frame,
                            text="Rectangle",
                            command=self.create_rectangle)
rect_button.pack(side=ctk.TOP, padx=5, pady=5)

oval_button = ctk.CTkButton(self.left_frame,
                           text="Oval",
                           command=self.create_oval)
oval_button.pack(side=ctk.TOP, padx=5, pady=5)

tri_button = ctk.CTkButton(self.left_frame,
                           text="Triangle",
                           command=self.create_triangle)
tri_button.pack(side=ctk.TOP, padx=5, pady=5)

def create_rectangle(self):
    rect_obj = Rectangle(self.canvas, 100, 100, 250, 250,
                         fill_color="red",
                         border_color="blue",
                         border_width=5)
    self.canvas.shape_list.append(rect_obj)
    self.canvas.draw_shapes()

def create_oval(self):
    oval_obj = Oval(self.canvas, 300, 100, 450, 250,
                   fill_color="red",
                   border_color="blue",
                   border_width=5)
    self.canvas.shape_list.append(oval_obj)
    self.canvas.draw_shapes()
```

```

def create_triangle(self):
    tri_obj = Triangle(self.canvas, 500, 100, 650, 250,
                       fill_color="red",
                       border_color="blue",
                       border_width=5)
    self.canvas.shape_list.append(tri_obj)
    self.canvas.draw_shapes()

def set_fill_color(self):
    if self.canvas.mouse.selected_obj:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.canvas.mouse.selected_obj.fill_color = color
        self.canvas.draw_shapes()

def set_border_color(self):
    if self.canvas.mouse.selected_obj:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.canvas.mouse.selected_obj.border_color = color
        self.canvas.draw_shapes()

def set_border_width(self, choice):
    if self.canvas.mouse.selected_obj:
        self.canvas.mouse.selected_obj.border_width = choice
        self.canvas.draw_shapes()

```

Refactored Mouse Class

```

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas

        self.selected_obj = None

    # Canvas mouse bindings
    self.canvas.bind('<Button-1>', self.left_mouse_down)
    self.canvas.bind('<Button-2>', self.middle_mouse_down)
    self.canvas.bind('<Button-3>', self.right_mouse_down)

    def left_mouse_down(self, event):
        if self.selected_obj:
            self.selected_obj.is_selected = False

```

```

        self.selected_obj = None
        self.canvas.draw_shapes()
    else:
        x, y = event.x, event.y
        self.select_shape(x, y)
        if self.selected_obj:
            # print("Object found at: ", self.selected_obj, event.x,
event.y)
            self.canvas.draw_shapes()

def middle_mouse_down(self, event):
    print("Middle click at ", event.x, event.y)

def right_mouse_down(self, event):
    print("Right click at ", event.x, event.y)

def select_shape(self, x, y):
    for s in self.canvas.shape_list:
        w = s.x2 - s.x1
        h = s.y2 - s.y1
        if (
            s.x1 <= x <= s.x2 + w
            and s.y1 <= y <= s.y2 + h
        ):
            self.unselect_all_objects()
            self.selected_obj = s
            self.selected_obj.is_selected = True

def unselect_all_objects(self):
    for s in self.canvas.shape_list:
        s.is_selected = False
    self.canvas.draw_shapes()

```

Mouse Bindings

There are many ways to handle mouse bindings for the canvas. Complex event handlers with if statements and flags become unwieldy quickly. I prefer to bind and unbind mouse event handlers based on the current mode of operation. This results in mouse bindings that are smaller and easier to manage. What do I mean by that?

The project will implement four modes of operation:

- Draw new shapes with mouse

- Select and move
- Rotation
- Resize with mouse

We can unbind a set of mouse events and then bind a new set of mouse events as needed. Create an unbind mouse events method in the mouse class.

```
class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas

        self.selected_obj = None

        # Canvas mouse bindings
        self.canvas.bind('<Button-1>', self.left_mouse_down)
        self.canvas.bind('<Button-2>', self.middle_mouse_down)
        self.canvas.bind('<Button-3>', self.right_mouse_down)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")
```

Draw New Shapes

Currently, shapes are created at a hard-coded positions on the computer screen when the shape button is selected. Lets add the capability to dynamically draw the shape using left mouse drag starting at any point on the canvas. This should allow us to create multiple copies of the same type of shape. Later we will add a shape move capability to move existing shapes to any point on the canvas.

Create the framework for the draw new shapes binding in the mouse class with bindings for left mouse down, left mouse drag (motion), and left mouse up.

```
def unbind_mouse_events(self):
    self.canvas.unbind("<Button-1>")
    self.canvas.unbind("<B1-Motion>")
    self.canvas.unbind("<ButtonRelease-1>")

def draw_bind_mouse_events(self):
```

```

    self.canvas.bind("<Button-1>", self.draw_left_down)
    self.canvas.bind("<B1-Motion>", self.draw_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

def draw_left_down(self, event):
    pass

def draw_left_drag(self, event):
    pass

def draw_left_up(self, event):
    pass

```

The framework does not implement the event handlers, a "pass" statement in a function simply causes a return from the function with no error. This is good practice during code development when you want to setup a structure or framework without implementing all the source code.

The draw left down method captures the starting x, y mouse coordinates, creates a shape object based on the current shape selection which is set in the shape selection menu in the left frame, adds the shape to the shape list, and redraws the shapes on the canvas.

```

def draw_left_down(self, event):
    self.start_x = event.x
    self.start_y = event.y

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(
            self.canvas, self.start_x, self.start_y, self.start_x,
            self.start_y,
            fill_color=None, border_color="black", border_width=3
        )

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)
        self.canvas.draw_shapes()

```

The draw left drag method sets the x1, y1 coordinates to the `start_x`, `start_y` coordinates captured in the draw left down method. It then sets the x2, y2 coordinates to the current mouse position coordinates. Finally, it draws all shapes on the canvas. Since

the current shapes are deleted, this has the effect of drawing the shape as the mouse is dragged.

```
def draw_left_drag(self, event):
    if self.current_shape_obj:
        x, y = event.x, event.y
        self.current_shape_obj.x1, self.current_shape_obj.y1 =
self.start_x, self.start_y
        self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
    self.canvas.draw_shapes()
```

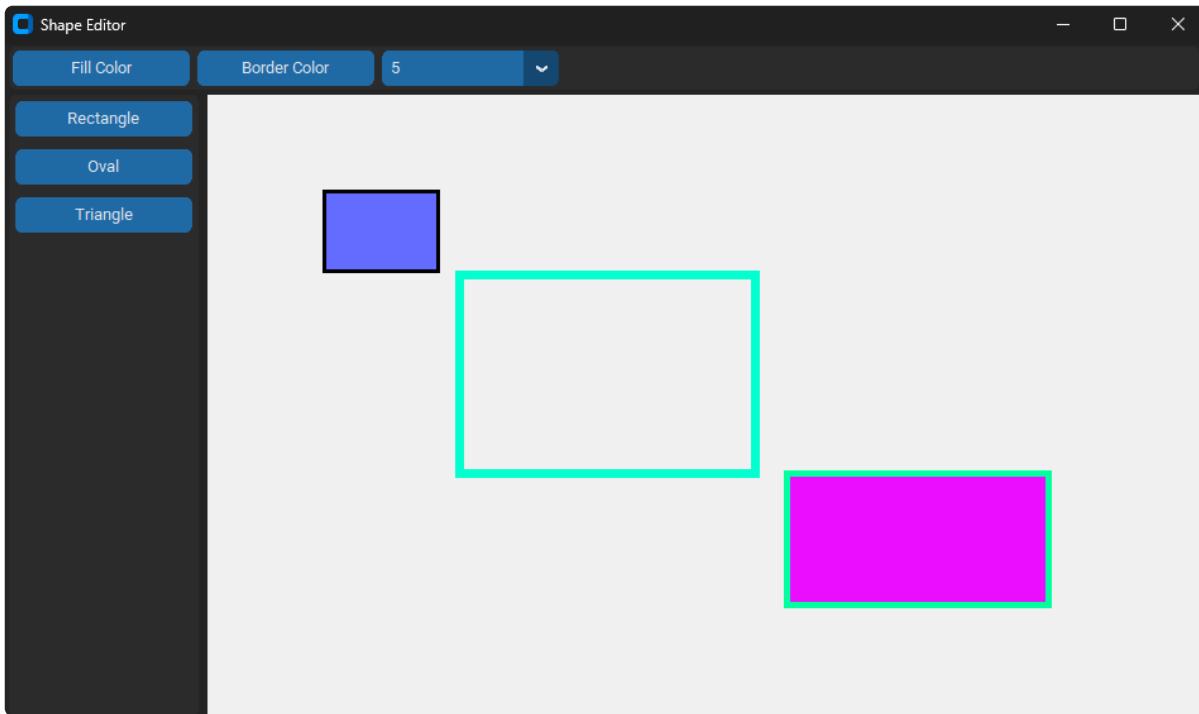
The draw left up method, resets the current shape and current shape obj variables, unbinds the draw mouse bindings and binds the left mouse down binding. This resets the bindings so that the new rectangle can be selected and the shape appearance controls work.

```
def draw_left_up(self, event):
    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.canvas.bind('<Button-1>', self.left_mouse_down)
```

Change the create rectangle function to set the current shape to "rectangle" and bind the draw mouse events.

```
def create_rectangle(self):
    self.canvas.mouse.current_shape = "rectangle"
    self.canvas.mouse.draw_bind_mouse_events()
```

Run the program, select the rectangle button, and draw rectangles using left mouse. Note that multiple rectangles can be drawn.



Implement the drag capability for ovals and triangles in the drag left down mouse event.

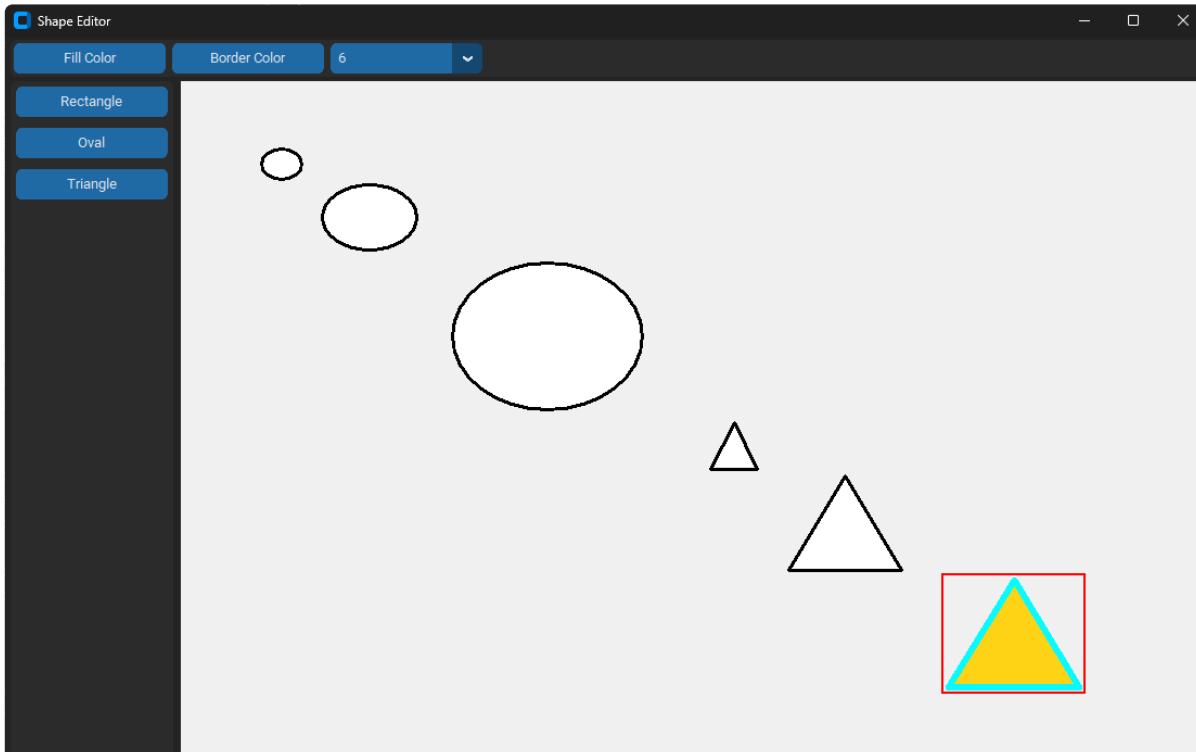
```
def draw_left_down(self, event):
    self.start_x = event.x
    self.start_y = event.y

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(
            self.canvas, self.start_x, self.start_y,
            self.start_x, self.start_y,
            fill_color="white",
            border_color="black",
            border_width=3)
    elif self.current_shape == "oval":
        self.current_shape_obj = Oval(self.canvas, self.start_x,
self.start_y,
            self.start_x, self.start_y,
            fill_color="white",
            border_color="black",
            border_width=3)
    elif self.current_shape == "triangle":
        self.current_shape_obj = Triangle(
            self.canvas, self.start_x, self.start_y,
            self.start_x + 100, self.start_y + 100,
            fill_color="white",
            border_color="black",
            border_width=3)
```

```
if self.current_shape_obj is not None:  
    self.canvas.shape_list.append(self.current_shape_obj)  
    self.canvas.draw_shapes()
```

Update the shape menu button drivers for oval and triangle.

```
def create_oval(self):  
    self.canvas.mouse.current_shape = "oval"  
    self.canvas.mouse.draw_bind_mouse_events()  
  
def create_triangle(self):  
    self.canvas.mouse.current_shape = "triangle"  
    self.canvas.mouse.draw_bind_mouse_events()
```



Not bad. We can now draw shapes of multiple sizes anywhere on the canvas and change the shape appearance.

Move Shapes

Create the framework for move mouse bindings. This should look familiar by now.

```

def bind_move_mouse_events(self):
    self.canvas.bind("<Button-1>", self.move_left_down)
    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

def move_left_down(self, event):
    pass

def move_left_drag(self, event):
    pass

def move_left_up(self, event):
    pass

```

The move event methods will perform two tasks: select the shape at the mouse position and move the shape with left mouse drag. Add offsets to the mouse class initializer.

```

self.start_x, self.start_y = 0, 0
self.offset_x1, self.offset_y1 = 0, 0
self.offset_x2, self.offset_y2 = 0, 0

```

Implement the move left down method which selects a shape at the mouse position and stores the offsets from the current shape position.

```

def move_left_down(self, event):
    x, y = event.x, event.y
    self.select_shape(x, y)
    if self.canvas.selected:
        x1, y1 = self.canvas.selected.x1, self.canvas.selected.y1
        x2, y2 = self.canvas.selected.x2, self.canvas.selected.y2
        self.offset_x1 = event.x - x1
        self.offset_y1 = event.y - y1
        self.offset_x2 = event.x - x2
        self.offset_y2 = event.y - y2
        self.canvas.draw_shapes()

```

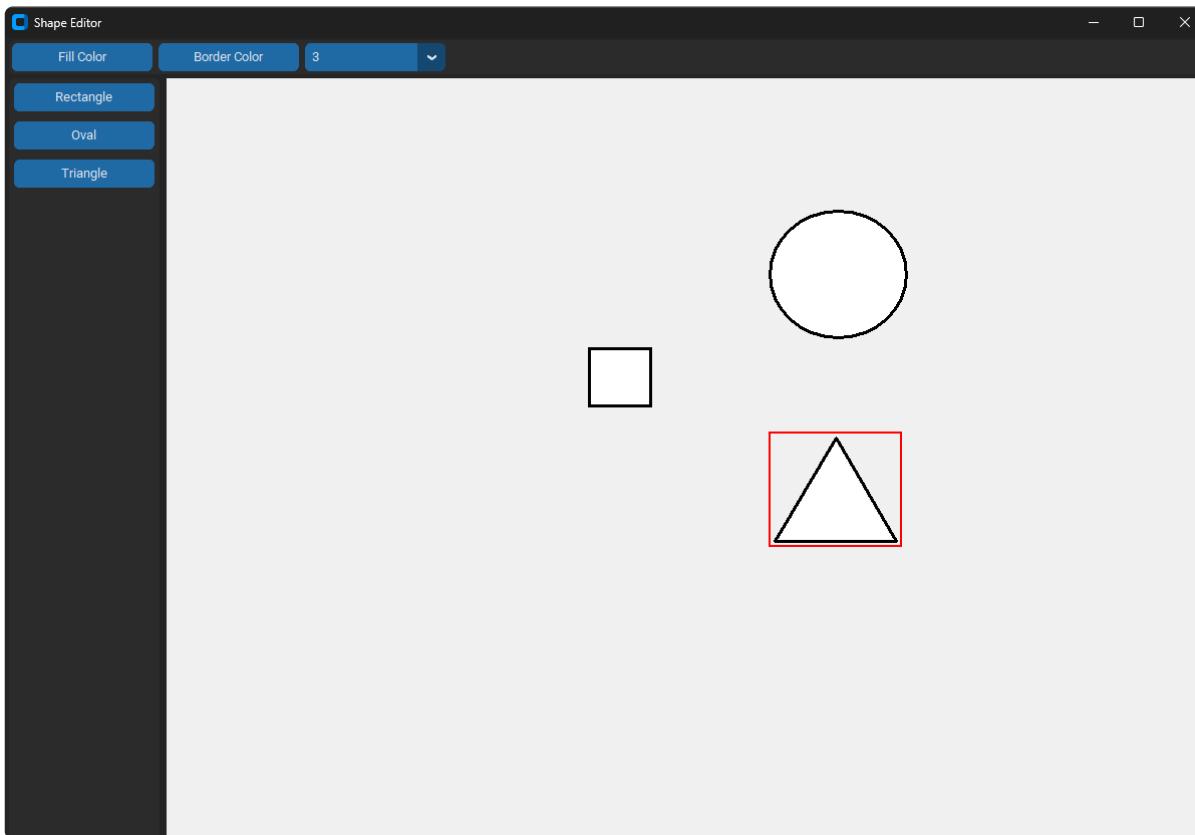
The move left drag method updates the shape coordinates with offset captured in the left mouse down method and redraws the canvas.

```
def move_left_drag(self, event):
    if self.canvas.selected:
        x = event.x - self.offset_x1
        y = event.y - self.offset_y1
        self.canvas.selected.x1, self.canvas.selected.y1 = x, y
        x = event.x - self.offset_x2
        y = event.y - self.offset_y2
        self.canvas.selected.x2, self.canvas.selected.y2 = x, y
    self.canvas.draw_shapes()
```

The move left up method resets the offset to 0.

```
def move_left_up(self, _event):
    self.offset_x1 = 0
    self.offset_y1 = 0
    self.offset_x2 = 0
    self.offset_y2 = 0
```

The shapes can now be moved with the left mouse button.



Shape Rotation

This application will rotate a shape by 90 degrees in a clockwise (cw) or counter-clockwise (ccw) direction. We will use keyboard bindings to rotate the currently selected shape where the 'r' key rotates ccw and the 'e' key rotates cw.

Add an "angle" variable to the Shape class so that all shapes can rotate.

```
class Shape:  
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,  
                 border_width):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.fill_color = fill_color  
        self.border_color = border_color  
        self.border_width = border_width  
  
        self.id = None  
        self.is_selected = False  
        self.angle = 0
```

Add bindings for the 'r' and 'e' keys to the Application just before the create rectangle method.

```
# Mouse & keyboard bindings  
self.bind('<r>', self.canvas.rotate_shape_ccw)  
self.bind('<e>', self.canvas.rotate_shape_cw)  
  
def create_rectangle(self):  
    self.canvas.mouse.current_shape = "rectangle"  
    self.canvas.mouse.draw_bind_mouse_events()
```

Note that these bindings call a rotation methods in the canvas class.

```
class Canvas(ctk.CTkCanvas):  
    def __init__(self):  
        super().__init__()  
  
        self.shape_list = []
```

```

    self.mouse = Mouse(self)
    self.selected = None

def draw_shapes(self):
    self.delete('all')
    for s in self.shape_list:
        s.draw()

def rotate_shape_ccw(self, _event):
    if self.selected is not None:
        self.selected.angle -= 90
        if self.selected.angle < 0:
            self.selected.angle = 270
    self.draw_shapes()

def rotate_shape_cw(self, _event):
    if self.selected is not None:
        self.selected.angle += 90
        if self.selected.angle > 270:
            self.selected.angle = 0
    self.draw_shapes()

```

Rectangle Rotation

Modify the Rectangle class to recalculate the shape coordinates based on the angle parameter before drawing the shape. We take advantage of the Rectangle symmetry because shape at angle=0 is the same at angle=180 and no rotation is needed. We do need to rotate the shape at angle=90 and angle=270. We do this by calculating the width and height and shape center. A new points list is added with the calculations for the rotated coordinates. The create rectangle method creates the rectangle using the points list. The selection rectangle is drawn with the corners offset from the current points list.

```

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color,
                        border_width)

    def draw(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = self.x1 + w/2, self.y1 + h/2
        if self.angle == 90 or self.angle == 180:
            points = [center[0] - h/2, center[1] - w/2,

```

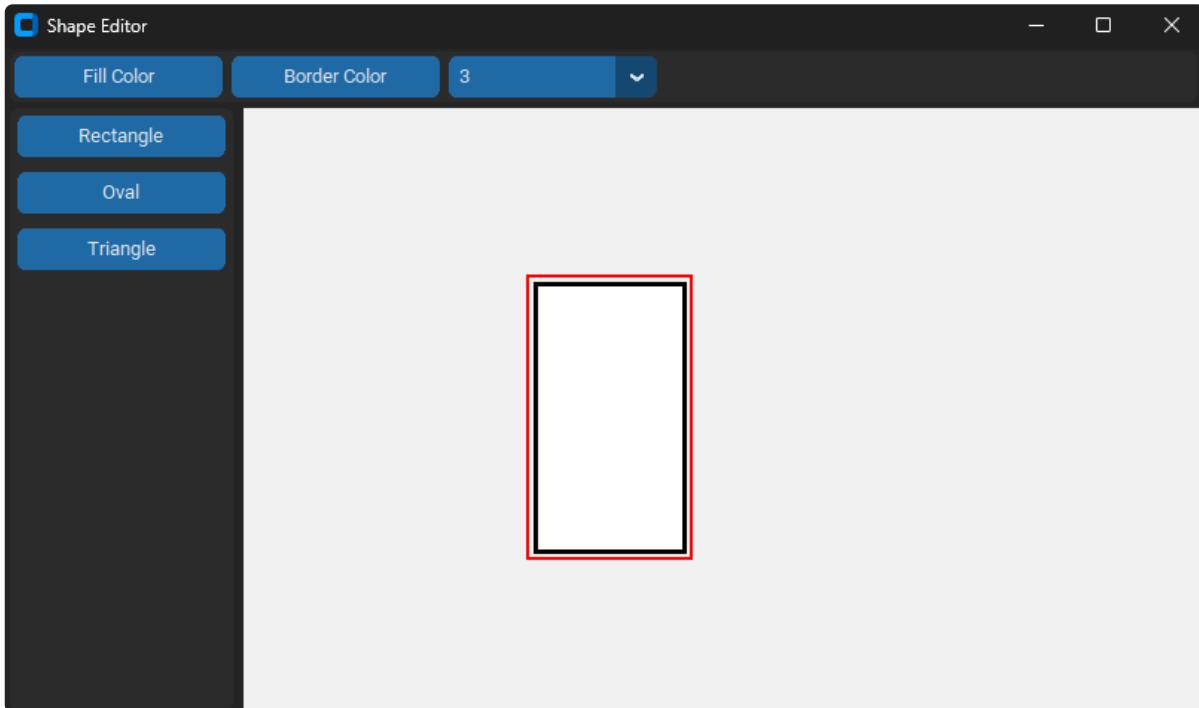
```
        center[0] + h/2, center[1] + w/2]

else:
    points = [self.x1, self.y1, self.x2, self.y2]

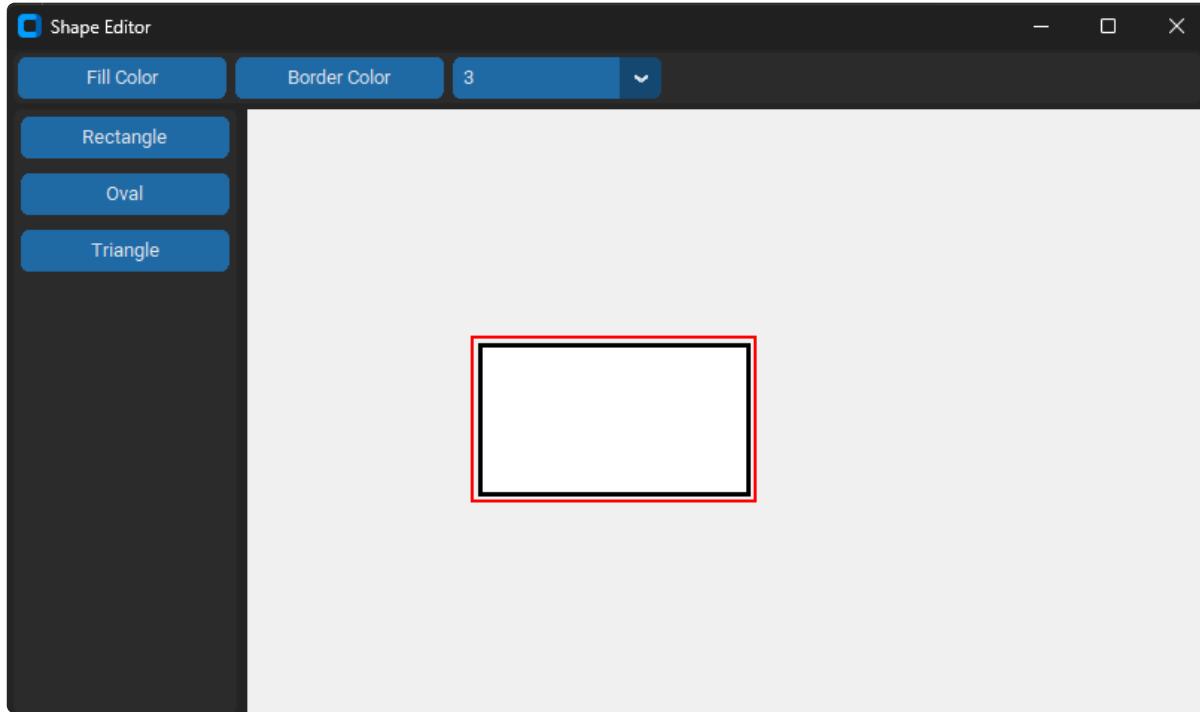
self.id = self.canvas.create_rectangle(points,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

if self.is_selected:
    sel_points = [points[0]-5, points[1]-5,
                  points[2]+5, points[3]+5]
    self.canvas.create_rectangle(sel_points, fill=None,
                                 outline="red", width=2)
```

Before hitting the 'r' key



After hitting the 'r' key



Oval Rotation

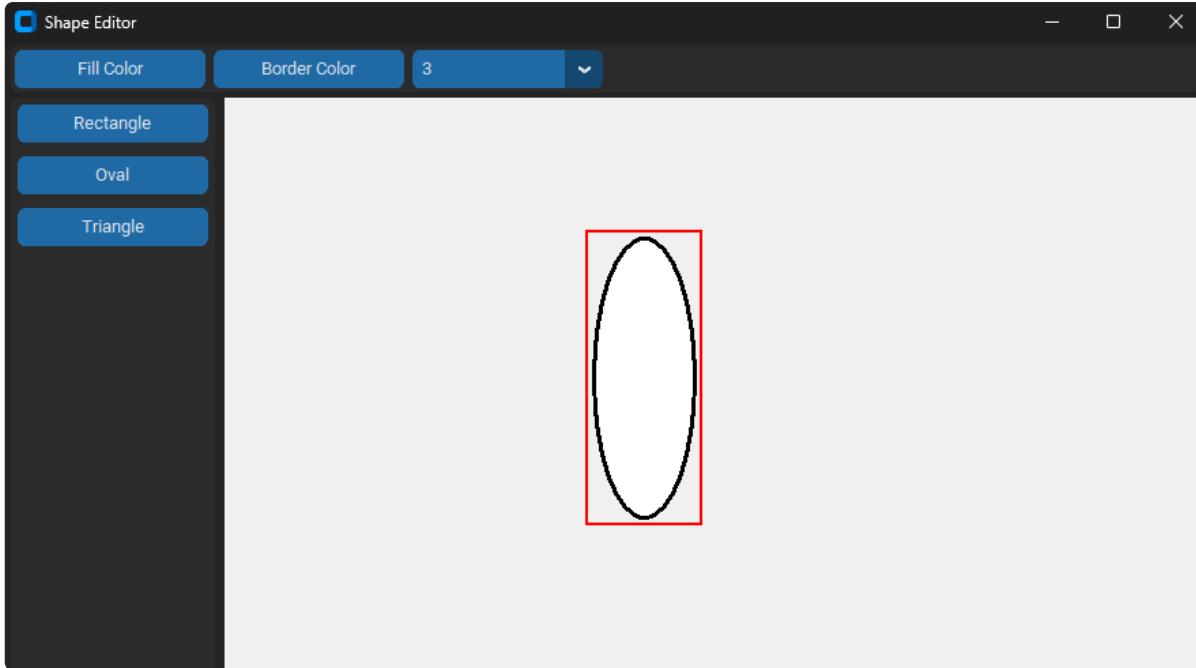
Modify the draw() method of Oval class using a similar pattern as that used with the Rectangle class to rotate the oval. Test using an ellipse to see the rotation.

```

if self.is_selected:
    sel_points = [points[0]-5, points[1]-5,
                  points[2]+5, points[3]+5]
    self.canvas.create_rectangle(sel_points, fill=None,
                                 outline="red", width=2)

```

After rotation image



Triangle Rotation

Rotation of non-symmetric polygon like a triangle is a bit trickier since the coordinates of the shape must be calculate for all four rotation angles.

```

class Triangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2,
                        fill_color, border_color, border_width)
        self.points = []

    def draw(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = self.x1 + w/2, self.y1 + h/2
        if self.angle == 0:
            x1, y1 = (self.x1, self.y1 + h)
            x2, y2 = (self.x1 + w, self.y1 + h)
            x3, y3 = (self.x1 + w / 2, self.y1)

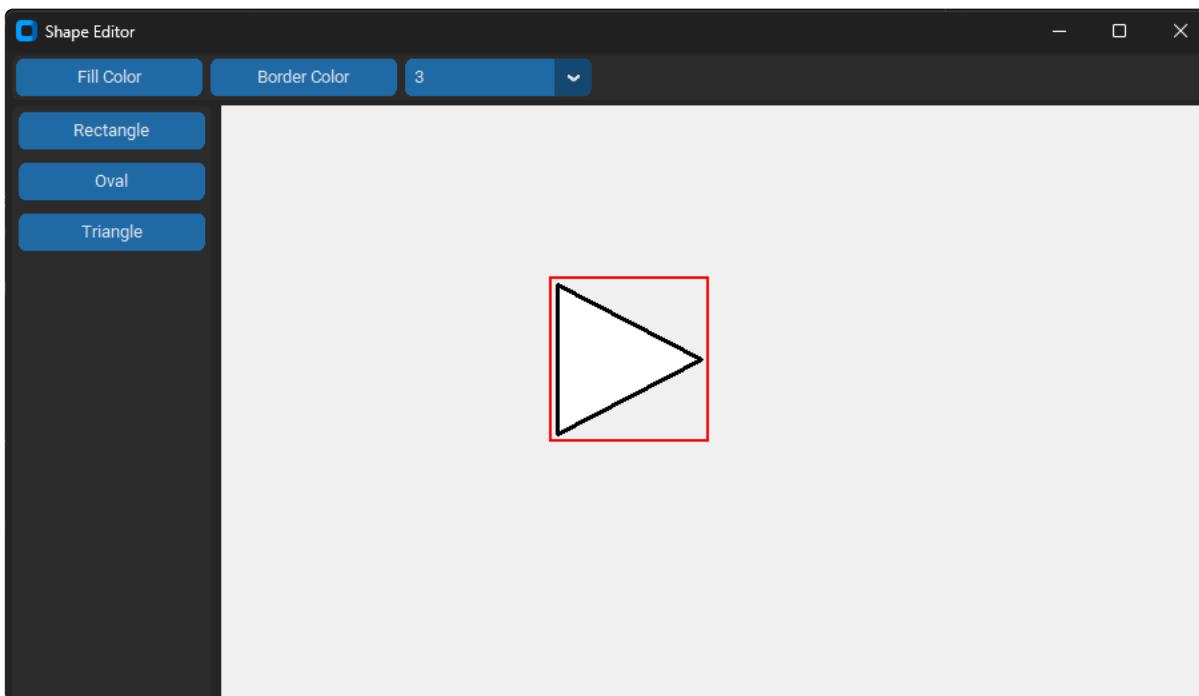
```

```

    elif self.angle == 90:
        x1, y1 = (center[0] - w / 2, center[1])
        x2, y2 = (center[0] + w / 2, center[1] + h / 2)
        x3, y3 = (center[0] + w / 2, center[1] - h / 2)
    elif self.angle == 180:
        x1, y1 = (center[0], center[1] + h/2)
        x2, y2 = (center[0] + w / 2, center[1] - h / 2)
        x3, y3 = (center[0] - w / 2, center[1] - h / 2)
    elif self.angle == 270:
        x1, y1 = (center[0] + w / 2, center[1])
        x2, y2 = (center[0] - w / 2, center[1] - h / 2)
        x3, y3 = (center[0] - w / 2, center[1] + h / 2)
    points = [x1, y1, x2, y2, x3, y3]
    self.id = self.canvas.create_polygon(points,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

if self.is_selected:
    sel_points = [center[0] - w/2 - 5, center[1] - h/2 - 5,
                  center[0] + w/2 + 5, center[1] + h/2 + 5]
    self.canvas.create_rectangle(sel_points, fill=None,
                                 outline="red", width=2)

```



Shape Resize

Shape resize can be implemented in a number of ways. For this application, let's bind the '+' and '-' keys to a scale factor of +10% and -10%, respectively.

Add an "scale" variable to the Shape class so that all shapes can rotate. We will assume a scale range of 0.1 (10%) to 10 (1000%) with default of 1 (100%).

```
class Shape:  
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,  
                 border_width):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.fill_color = fill_color  
        self.border_color = border_color  
        self.border_width = border_width  
  
        self.id = None  
        self.is_selected = False  
        self.angle = 0  
        self.scale = 1 # Default is 100% scale factor, range 0.1 to 10.0
```

Bind the '+' and '-' keys to two new canvas methods: scale up and scale down.

```
# Mouse & keyboard bindings  
self.bind('<r>', self.canvas.rotate_shape_ccw)  
self.bind('<e>', self.canvas.rotate_shape_cw)  
self.bind('<+>', self.canvas.scale_up)  
self.bind('<->', self.canvas.scale_down)
```

Add the two new methods to the canvas class.

```
class Canvas(ctk.CTkCanvas):  
    def __init__(self):  
        super().__init__()  
  
        self.shape_list = []  
        self.mouse = Mouse(self)  
        self.selected = None
```

```

def draw_shapes(self):
    self.delete('all')
    for s in self.shape_list:
        s.draw()

def rotate_shape_ccw(self, _event):
    if self.selected is not None:
        self.selected.angle -= 90
        if self.selected.angle < 0:
            self.selected.angle = 270
    self.draw_shapes()

def rotate_shape_cw(self, _event):
    if self.selected is not None:
        self.selected.angle += 90
        if self.selected.angle > 270:
            self.selected.angle = 0
    self.draw_shapes()

def scale_up(self, _event):
    if self.selected.scale < 10:
        self.selected.scale += 0.1

def scale_down(self, _event):
    if self.selected.scale > 0.1:
        self.selected.scale -= 0.1

```

To up scale a shape, increase its width and height by 10%. To down scale a shape, decrease its width and height by 10%.

Rectangle Resize

In the rectangle draw() method, multiply the width and height by the scale factor. Calculate the rotated and unrotated points based on the center point which is calculated after the width and height scaling.

```

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2, fill_color,
                        border_color, border_width)

    def draw(self):

```

```

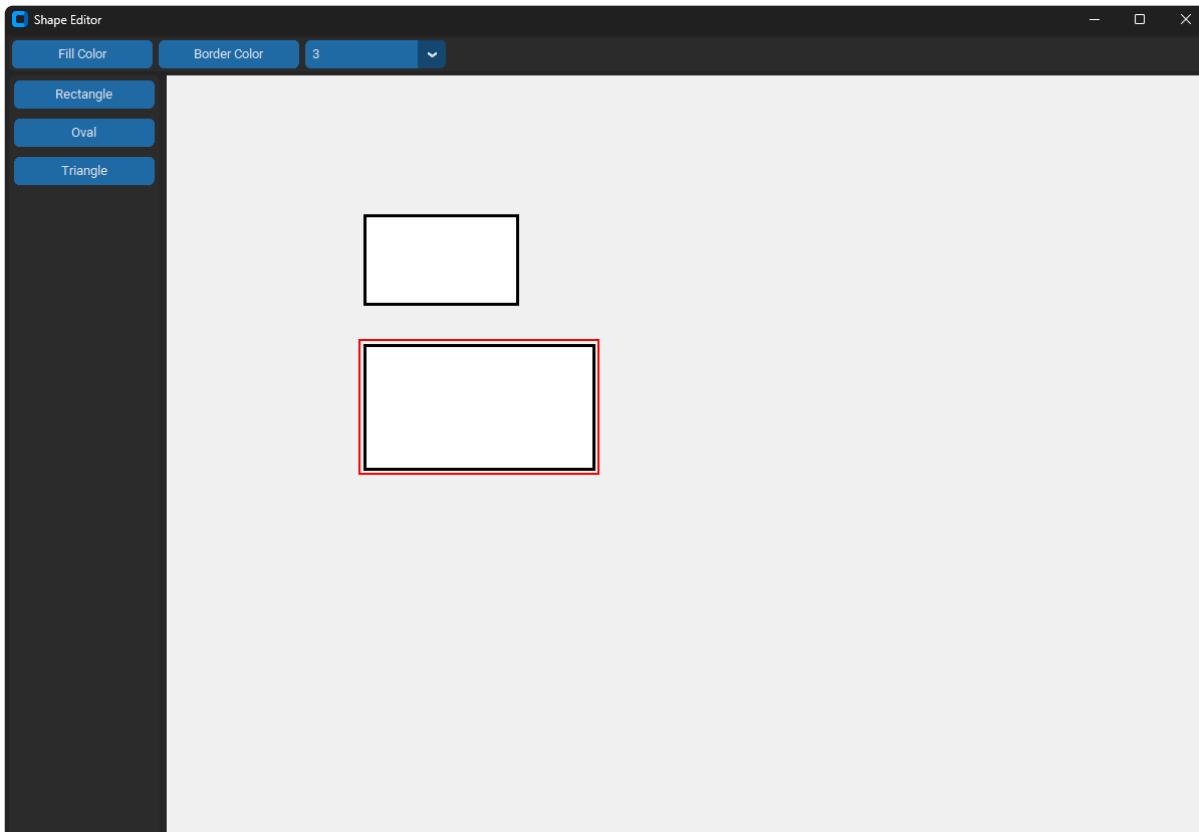
w, h = self.x2 - self.x1, self.y2 - self.y1
w = w * self.scale
h = h * self.scale
center = self.x1 + w/2, self.y1 + h/2
if self.angle == 90 or self.angle == 180:
    points = [center[0] - h/2, center[1] - w/2,
              center[0] + h/2, center[1] + w/2]
else:
    points = [center[0] - w/2, center[1] - h/2,
              center[0] + w/2, center[1] + h/2]

self.id = self.canvas.create_rectangle(points,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

if self.is_selected:
    sel_points = [points[0]-5, points[1]-5,
                  points[2]+5, points[3]+5]
    self.canvas.create_rectangle(sel_points, fill=None,
                                 outline="red", width=2)

```

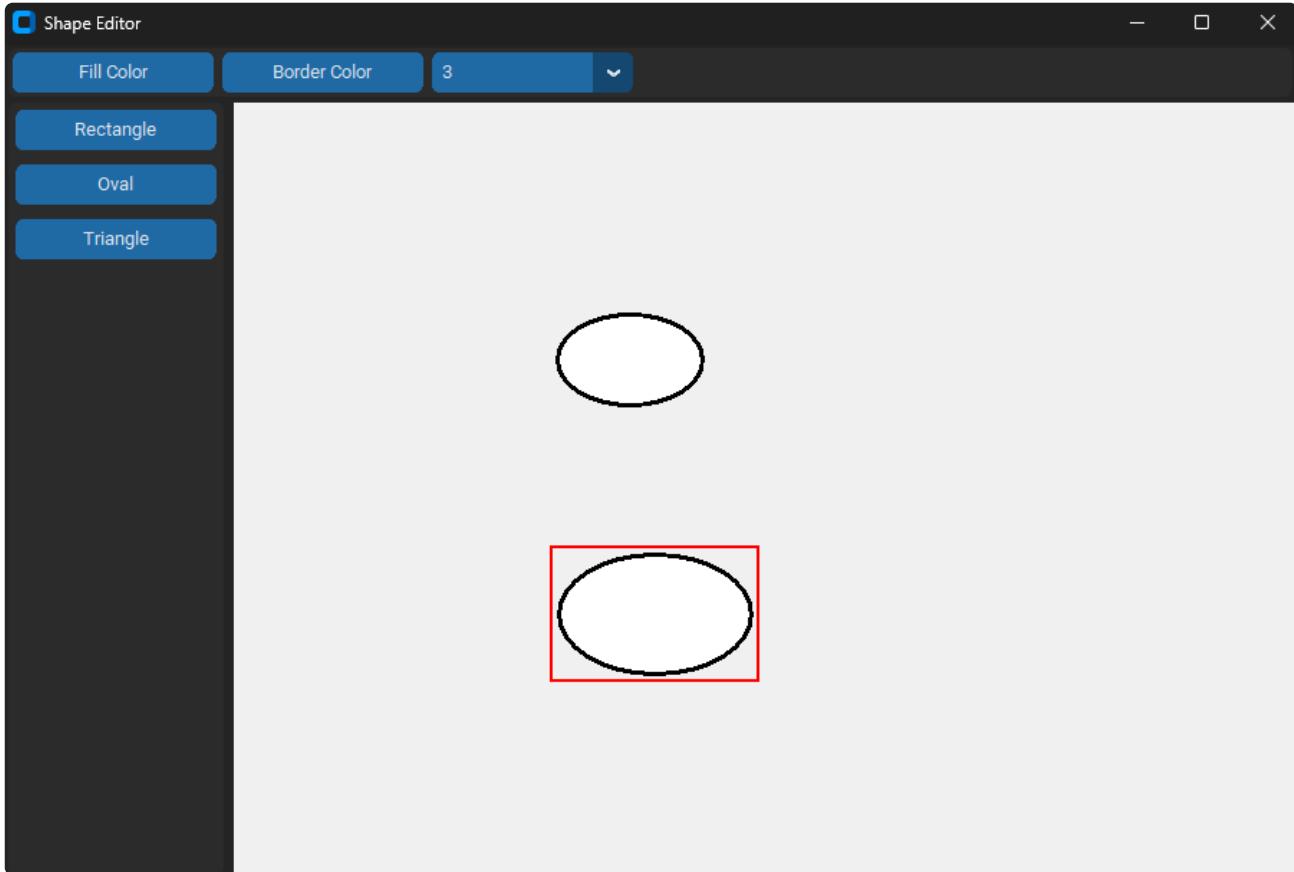
To test rectangle resize, run the program and draw two rectangle approximately the same size. Select one of the rectangles and press the '+' key several times to see the rectangle increase in size with each key press.



Oval Resize

Add the width and height scaling to the Oval draw() method using the same pattern as the Rectangle scaling.

Oval resize (scaling) test



Triangle resize

In the Triangle draw() method, we are already calculating all four rotation positions from the center point except at angle = 0. Add the width and height scaling and calculate angle=0 from the center point.

```
class Triangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2,
                         fill_color, border_color, border_width)
        self.points = []

    def draw(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        w = w * self.scale
        h = h * self.scale
        center = self.x1 + w/2, self.y1 + h/2
        if self.angle == 0:
            x1, y1 = (center[0], center[1] - h/2)
            x2, y2 = (center[0] + w / 2, center[1] + h / 2)
```

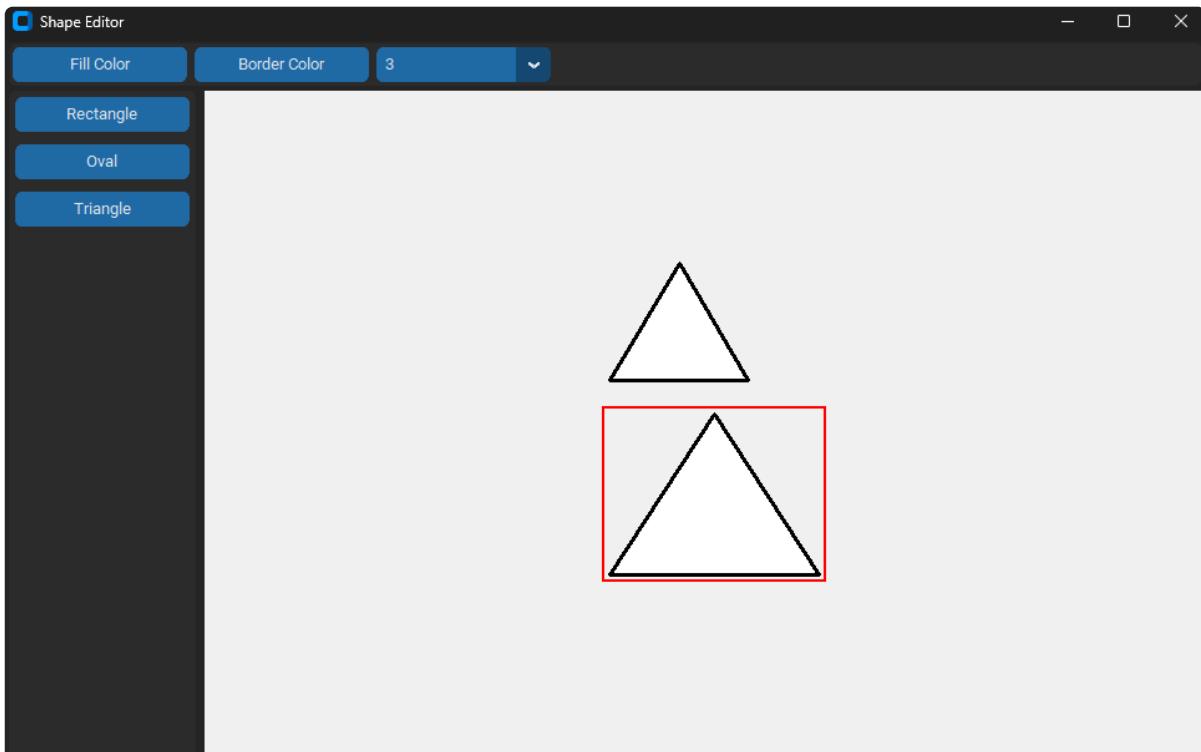
```

        x3, y3 = (center[0] - w / 2, center[1] + h / 2)
    elif self.angle == 90:
        x1, y1 = (center[0] - w / 2, center[1])
        x2, y2 = (center[0] + w / 2, center[1] + h / 2)
        x3, y3 = (center[0] + w / 2, center[1] - h / 2)
    elif self.angle == 180:
        x1, y1 = (center[0], center[1] + h/2)
        x2, y2 = (center[0] + w / 2, center[1] - h / 2)
        x3, y3 = (center[0] - w / 2, center[1] - h / 2)
    elif self.angle == 270:
        x1, y1 = (center[0] + w / 2, center[1])
        x2, y2 = (center[0] - w / 2, center[1] - h / 2)
        x3, y3 = (center[0] - w / 2, center[1] + h / 2)
    points = [x1, y1, x2, y2, x3, y3]
    self.id = self.canvas.create_polygon(points,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

    if self.is_selected:
        sel_points = [center[0] - w/2 - 5, center[1] - h/2 - 5,
                      center[0] + w/2 + 5, center[1] + h/2 + 5]
        self.canvas.create_rectangle(sel_points, fill=None,
                                    outline="red", width=2)

```

Test Triangle resize (scaling)



As an exercise left to the reader, use a difference method for resize where each shape displays a set of resize selectors at the four corners of the selection box. When the user clicks and drags on a resize/reshape selector, the shape changes size and shape based on the position of the dragged selector.

Grid

Most diagram editors like Microsoft Visio have a background grid and a snap-to-grid system. The grid can have the following features:

- Adjustable grid spacing in a range of 5 to 100 in steps of 5
- Grid on/off
- Shape snap-to-grid during draw and move events
- Snap-to-grid on/off
- Grid lines or dots

Grid class

```
class Grid:  
    type = "line"  
  
    def __init__(self, canvas, grid_size):  
        self.canvas = canvas  
        self.grid_size = grid_size  
        self.grid_visible = True  
        self.dash_list = None  
  
    def draw_grid(self):  
        if self.grid_visible:  
            w = self.canvas.winfo_width() # Get current width of canvas  
            h = self.canvas.winfo_height() # Get current height of canvas  
            self.canvas.delete('grid_line')  
  
            if Grid.type == "dot":  
                self.dash_list = [1, 1]  
            elif Grid.type == "line":  
                self.dash_list = None  
  
            # Creates all vertical lines at intervals of 100  
            for i in range(0, w, self.grid_size):  
                self.canvas.create_line([(i, 0), (i, h)], dash=self.dash_list,
```

```

        fill="#cccccc", tag='grid_line')

# Creates all horizontal lines at intervals of 100
for i in range(0, h, self.grid_size):
    self.canvas.create_line([(0, i), (w, i)], dash=self.dash_list,
        fill="#cccccc", tag='grid_line')

```

The grid class draws lines or dots in the vertical and horizontal direction at the grid size spacing. It checks to see if the grid is set to visible. Note that it deletes all previous grids using a "tag" in a canvas delete method. It also checks to see if the grid is set to "line" or "dot". The canvas create line method has a option to set "dash". By setting dash=[1,1], the line is drawn as a dotted line.

To persist the background grid if the window size is changed, bind "<Configure>" to a window resize handler and redraw all shapes including the grid.

```

# Mouse & keyboard bindings
self.bind('<r>', self.canvas.rotate_shape_ccw)
self.bind('<e>', self.canvas.rotate_shape_cw)
self.bind('+', self.canvas.scale_up)
self.bind('-', self.canvas.scale_down)
self.bind("<Configure>", self.on_window_resize)

def on_window_resize(self, _event):
    self.canvas.draw_shapes()

```

Update the canvas draw shapes method to draw the grid in the background. In the Canvas class initializer, set the grid size, create a Grid object call `self.grid`, and draw the grid.

```

class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()

        self.shape_list = []
        self.mouse = Mouse(self)
        self.selected = None

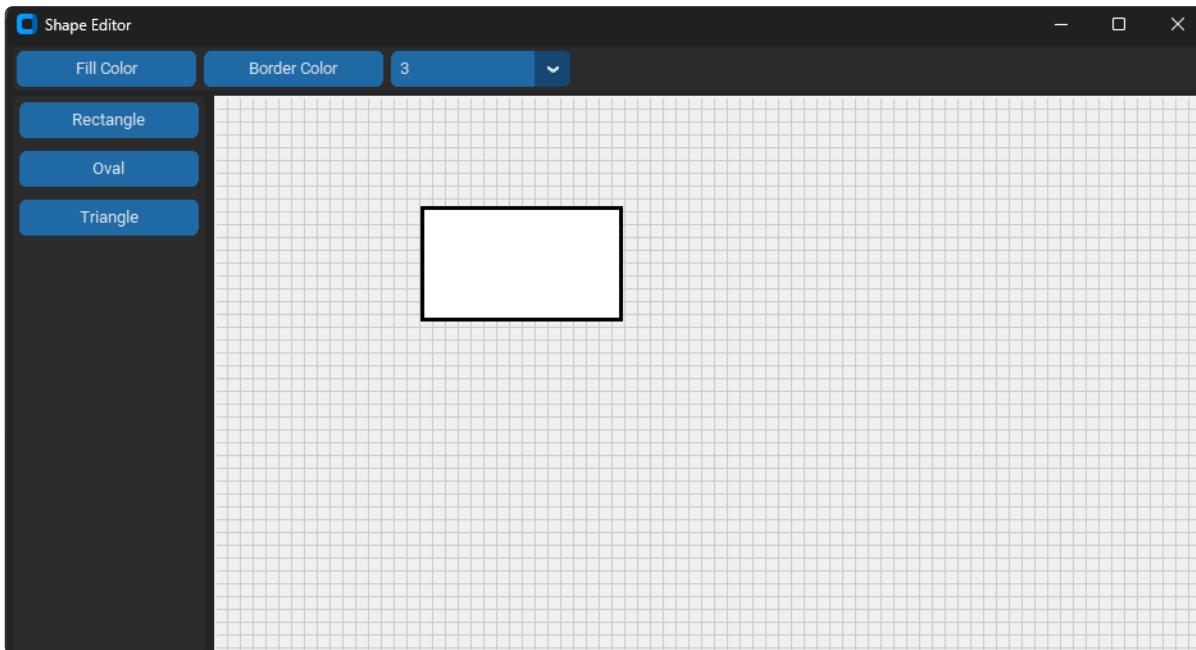
        self.grid_size = 10
        self.grid = Grid(self, self.grid_size)
        self.grid.draw_grid()

```

Update the Canvas draw_shapes method to draw the grid. Note the grid is drawn before the shapes so that it is in the background.

```
def draw_shapes(self):
    self.delete('all')
    self.grid.draw_grid()
    for s in self.shape_list:
        s.draw()
```

Run the program to verify that the grid is drawn and that it properly resizes when the window size changes.



Shape Snap-to-Grid

Create a `snap_to_grid` function that given a set of x, y coordinates, it calculates the nearest x, y point on the grid and returns those coordinates.

For example, in a mouse down event

```
x = event.x
y = event.y
x, y = snap_to_grid(x, y)
```

where

```
def snap_to_grid(x, y):
    x = round(x / grid_size) * grid_size
    y = round(y / grid_size) * grid_size
    return x, y
```

Note that the Python "round" function will round the given number to the nearest integer. Since this function is associated with the grid, add a new method to the Grid class. Add a test to see if the grid is visible

```
# Creates all horizontal lines at intervals of 100
for i in range(0, h, self.grid_size):
    self.canvas.create_line([(0, i), (w, i)], dash=self.dash_list,
                           fill='#cccccc', tag='grid_line')

def snap_to_grid(self, x, y):
    if self.grid_visible:
        x = round(x / self.grid_size) * self.grid_size
        y = round(y / self.grid_size) * self.grid_size
    return x, y
```

Implement `snap_to_grid` in the mouse draw event handler and the mouse move event handler.

Mouse draw event handlers

```
def draw_left_down(self, event):
    self.start_x = event.x
    self.start_y = event.y
    self.start_x, self.start_y = \
        (self.canvas.grid.snap_to_grid(self.start_x, self.start_y))

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas,
            self.start_x, self.start_y,
            self.start_x, self.start_y,
            fill_color="white",
            border_color="black",
            border_width=3)
    elif self.current_shape == "oval":
        self.current_shape_obj = Oval(self.canvas,
            self.start_x, self.start_y,
```

```

        self.start_x, self.start_y,
        fill_color="white",
        border_color="black",
        border_width=3)
    elif self.current_shape == "triangle":
        self.current_shape_obj = Triangle(self.canvas,
            self.start_x, self.start_y,
            self.start_x + 100, self.start_y + 100,
            fill_color="white",
            border_color="black",
            border_width=3)

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)
        self.canvas.draw_shapes()

def draw_left_drag(self, event):
    if self.current_shape_obj:
        x, y = event.x, event.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.current_shape_obj.x1, self.current_shape_obj.y1 = self.start_x,
self.start_y
        self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
        self.canvas.draw_shapes()

def draw_left_up(self, event):
    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.bind_move_mouse_events()

```

Mouse move event handlers

```

def move_left_down(self, event):
    x, y = event.x, event.y
    x, y = self.canvas.grid.snap_to_grid(x, y)
    self.select_shape(x, y)
    if self.canvas.selected:
        x1, y1 = self.canvas.selected.x1, self.canvas.selected.y1
        x2, y2 = self.canvas.selected.x2, self.canvas.selected.y2
        self.offset_x1 = event.x - x1
        self.offset_y1 = event.y - y1
        self.offset_x2 = event.x - x2
        self.offset_y2 = event.y - y2

```

```

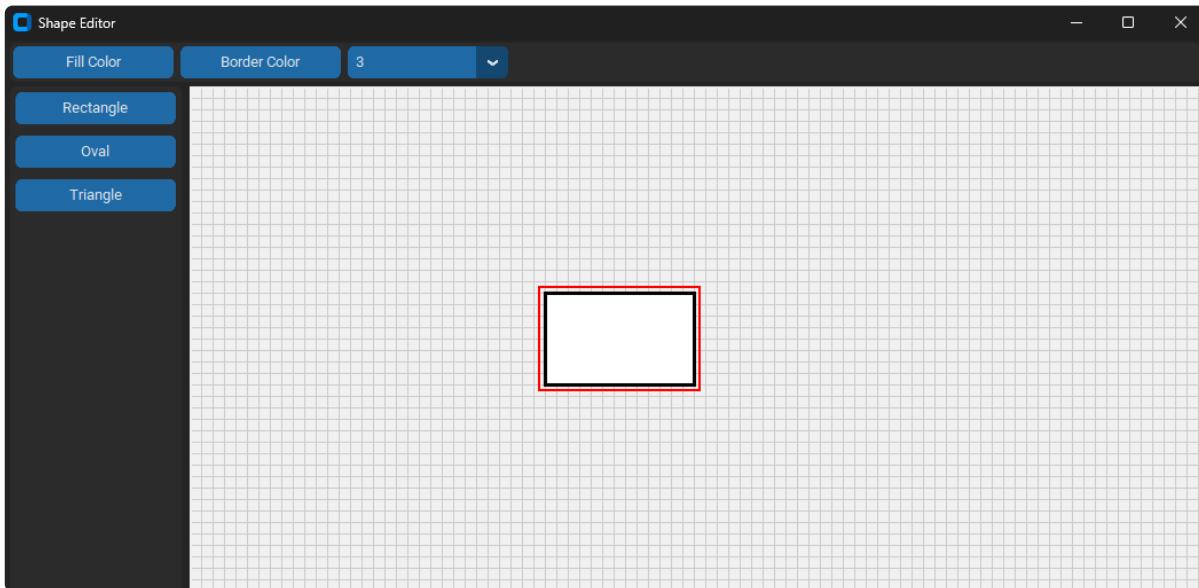
    self.canvas.draw_shapes()

def move_left_drag(self, event):
    if self.canvas.selected:
        x = event.x - self.offset_x1
        y = event.y - self.offset_y1
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.canvas.selected.x1, self.canvas.selected.y1 = x, y
        x = event.x - self.offset_x2
        y = event.y - self.offset_y2
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.canvas.selected.x2, self.canvas.selected.y2 = x, y
    self.canvas.draw_shapes()

def move_left_up(self, _event):
    self.offset_x1 = 0
    self.offset_y1 = 0
    self.offset_x2 = 0
    self.offset_y2 = 0

```

Run the program and verify that shape draw and move operations snap to the grid.



Grid Visibility and Size

Add a [ctk switch control](#) to the top frame to control grid visibility. Add an option menu to the top frame to control grid size.

```

# Add top frame widgets here
self.fill_color_button = ctk.CTkButton(self.top_frame,

```

```

                text="Fill Color",
                command=self.set_fill_color)
self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

self.border_color_button = ctk.CTkButton(self.top_frame,
                                         text="Border Color",
                                         command=self.set_border_color)
self.border_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

border_width_optionmenu = ctk.CTkOptionMenu(self.top_frame,
                                             values=["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"],
                                             command=self.set_border_width)
border_width_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
border_width_optionmenu.set("3")

self.switch_var = ctk.StringVar(value="on")
switch = ctk.CTkSwitch(self.top_frame, text="Grid Visible",
                       command=self.grid_visibility,
                       variable=self.switch_var,
                       onvalue="on", offvalue="off")
switch.pack(side=ctk.LEFT, padx=3, pady=3)

grid_size_optionmenu = ctk.CTkOptionMenu(self.top_frame,
                                         values=["5", "10", "15", "20", "25", "40", "50",
                                                 "60", "70", "80", "90", "100"],
                                         command=self.set_grid_size)
grid_size_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
grid_size_optionmenu.set("10")

```

Add the switch grid visibility method and the option menu set grid size method.

```

def grid_visibility(self):
    if self.switch_var.get() == "on":
        self.canvas.grid.grid_visible = True
    elif self.switch_var.get() == "off":
        self.canvas.grid.grid_visible = False
    self.canvas.draw_shapes()

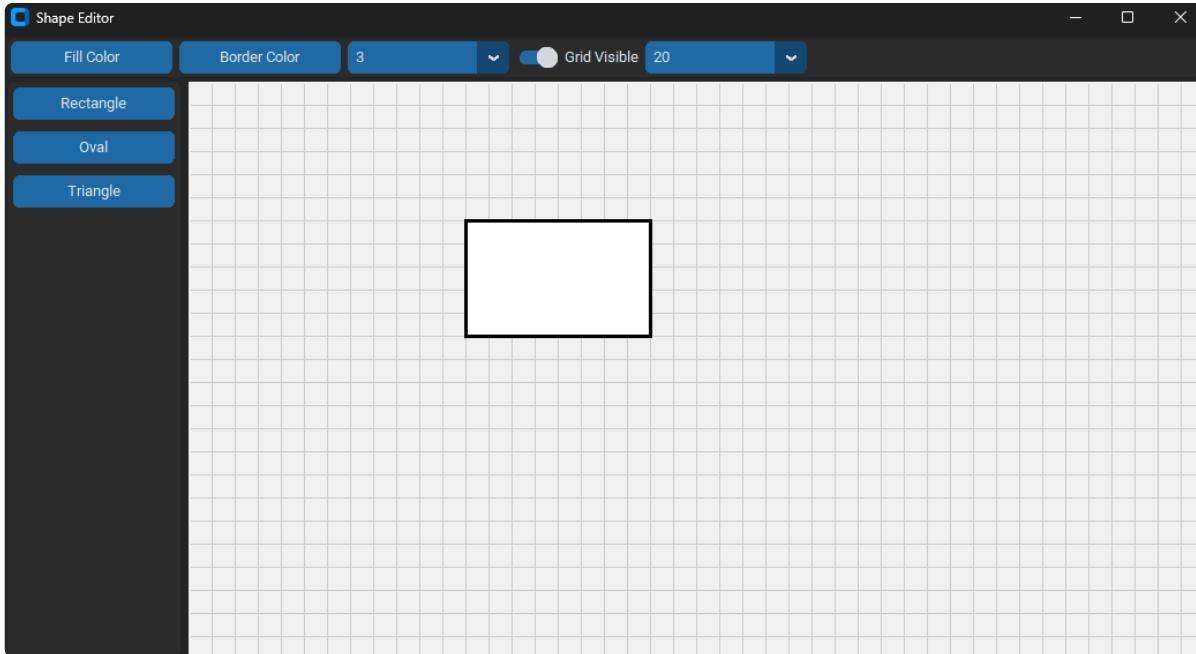
def set_grid_size(self, choice):
    self.canvas.grid.grid_size = int(choice)
    self.canvas.draw_shapes()

if __name__ == "__main__":

```

```
"""Instantiate the Shape Editor application and run the main loop"""
app = ShapeEditorApp()
app.mainloop()
```

Test grid visibility on/off and verify that snap to grid is off when the grid is off. Test the grid size control.



Modularization

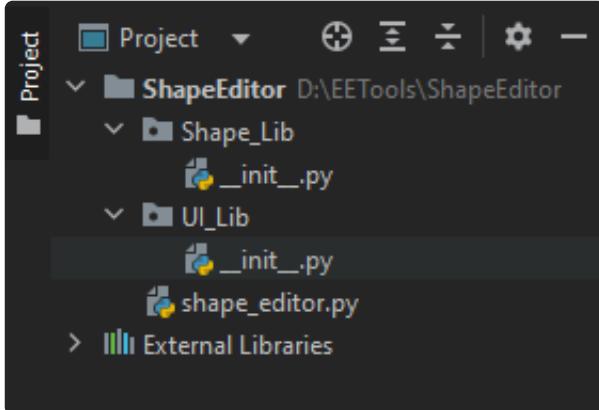
The application now has 460 lines of code. It is time to refactor it to creates smaller, manageable modules and packages.

A python module is a file that contains code that can be imported into the an application. The module code is usually a class or set of classes.

A python package is a subdirectory in the main project directory that contains modules and an empty `__init__.py` file. The `__init__.py` file identifies the directory as a package. I usually leave the file empty although it can contain code to modify its import style.

Create two directories under the ShapeEditor called `UI_Lib` and `Shape_Lib`. Add empty `__init__.py` files to both directories. Note that the icons for the directories in Project tab change when the directory becomes a package.

Directory structure



Start with the shape classes and create a separate file for each class:

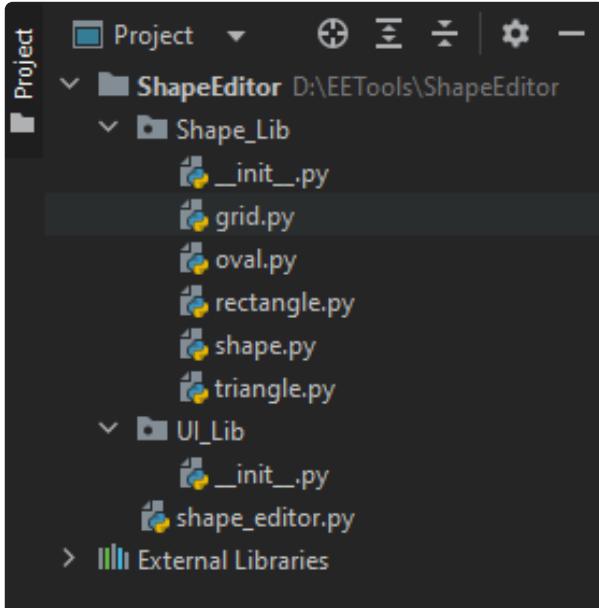
- Shape class
- Rectangle class
- Oval class
- Triangle class
- Grid class

We need to tell the Rectangle, Oval, and Triangle classes where to find the Shape class.

Add the following line to the top of each sub-class:

```
from Shape_Lib.shape import Shape
```

Directory structure after modularization of the shape classes



Delete the shape classes from the `shape_editor.py` file and add import statements for the shape modules.

```
import customtkinter as ctk
from CTkColorPicker import *

from Shape_Lib.rectangle import Rectangle
from Shape_Lib.oval import Oval
from Shape_Lib.triangle import Triangle
from Shape_Lib.grid import Grid
```

Run the program and verify that it runs as normal. The number of lines of code in `shape_editor.py` has been reduced to 309 lines.

Next move UI classes to the `UI_Lib` directory

- Canvas class
- Mouse class

Mouse Module

Cut the mouse class code from `shape_editor.py` and paste it into the mouse module called `mouse.py`. The mouse module needs access to the shape classes for Rectangle, Oval, and Triangle. Add imports for the shape classes at the top of the mouse module.

```
from Shape_Lib.rectangle import Rectangle
from Shape_Lib.oval import Oval
from Shape_Lib.triangle import Triangle

class Mouse:
```

Canvas Module

Cut the canvas class code from `shape_editor.py` and paste it into the canvas module called `canvas.py`. Add imports at the top of the canvas module for `customtkinter`, `Mouse`, and `Grid`.

```
import customtkinter as ctk
from UI_Lib.mouse import Mouse
from Shape_Lib.grid import Grid

class Canvas(ctk.CTkCanvas):
```

Modify `shape_editor.py` to import the canvas module which imports the mouse module. You can delete the unused shape imports since they moved to the mouse and canvas modules.

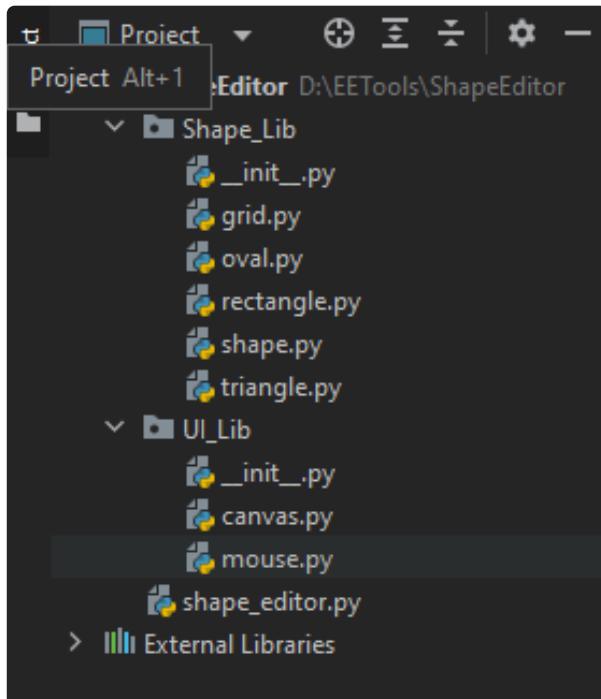
```
import customtkinter as ctk
from CTkColorPicker import *

from UI_Lib.canvas import Canvas

class ShapeEditorApp(ctk.CTk):
```

Run the program and verify that it runs without error.

Directory structure



`shape_editor.py` now has 141 lines of code.

Further modularization can be achieved by modularizing the Top Frame and Left Frame in to modules where the frames inherit from CTk Frame class.

Top Frame Module

Create a new file in the UI_Lib directory called `top_frame.py`. Create a class called `TopFrame` which inherits from `ctk.CTkFrame`.

```
import customtkinter as ctk
from CTkColorPicker import *

class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
```

The frame classes need a parent parameter which is the parent container and a canvas parameter.

Cut the Top Frame code from `shape_editor.py` and paste it into the body of the Top Frame module. Add the control method handlers associated with the top frame controls. Change references to `self.top_frame` to `self` since the class is the top frame. Some modification to the selected obj variable name is needed for the newly modularized code.

```
import customtkinter as ctk
from CTkColorPicker import *

class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add top frame widgets here
        self.fill_color_button = ctk.CTkButton(self,
                                              text="Fill Color",
                                              command=self.set_fill_color)
        self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)
```

```

        self.border_color_button = ctk.CTkButton(self,
                                                text="Border Color",
                                                command=self.set_border_color)
        self.border_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

    border_width_optionmenu = ctk.CTkOptionMenu(self,
                                                values=[ "1", "2", "3", "4", "5", "6", "7", "8", "9", "10" ],
                                                command=self.set_border_width)
    border_width_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
    border_width_optionmenu.set("3")

    self.switch_var = ctk.StringVar(value="on")
    switch = ctk.CTkSwitch(self, text="Grid Visible",
                           command=self.grid_visibility,
                           variable=self.switch_var,
                           onvalue="on", offvalue="off")
    switch.pack(side=ctk.LEFT, padx=3, pady=3)

    grid_size_optionmenu = ctk.CTkOptionMenu(self,
                                              values=[ "5", "10", "15", "20", "25", "40", "50",
                                                       "60", "70", "80", "90", "100" ],
                                              command=self.set_grid_size)
    grid_size_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
    grid_size_optionmenu.set("10")

def set_fill_color(self):
    if self.canvas.selected:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.canvas.selected.fill_color = color
        self.canvas.draw_shapes()

def set_border_color(self):
    if self.canvas.selected:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.canvas.selected.border_color = color
        self.canvas.draw_shapes()

def set_border_width(self, choice):
    if self.canvas.selected:
        self.canvas.selected.border_width = choice
        self.canvas.draw_shapes()

def grid_visibility(self):

```

```

if self.switch_var.get() == "on":
    self.canvas.grid.grid_visible = True
elif self.switch_var.get() == "off":
    self.canvas.grid.grid_visible = False
self.canvas.draw_shapes()

def set_grid_size(self, choice):
    self.canvas.grid.grid_size = int(choice)
    self.canvas.draw_shapes()

```

In `shape_editor.py` add import for the Top Frame class and modify the top frame creation to use the custom Top Frame class.

```

import customtkinter as ctk

from UI_Lib.canvas import Canvas
from UI_Lib.top_frame import TopFrame


class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""

    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        self.shape_list = []

        # Add widgets to the app here
        self.canvas = Canvas()
        self.top_frame = TopFrame(self, self.canvas)
        self.left_frame = ctk.CTkFrame(master=self)

```

Run the program and verify that the top frame controls function as normal.

Left Frame Module

Similarly, modularize the Left Frame by creating `left_frame.py` in `UI_Lib`. Create a class called `LeftFrame` with the `ctk.CTkFrame` as the base class.

```
import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
```

Cut the left frame widgets from `shape_editor.py` and paste in the Top Frame module. Change references from `self.left_frame` to `self`. Cut the shape button handlers also and paste them into the Top Frame Module.

```
import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self,
                                    text="Rectangle",
                                    command=self.create_rectangle)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

        oval_button = ctk.CTkButton(self,
                                    text="Oval",
                                    command=self.create_oval)
        oval_button.pack(side=ctk.TOP, padx=5, pady=5)

        tri_button = ctk.CTkButton(self,
                                    text="Triangle",
                                    command=self.create_triangle)
        tri_button.pack(side=ctk.TOP, padx=5, pady=5)

    def create_rectangle(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.canvas.mouse.draw_bind_mouse_events()
```

```

def create_oval(self):
    self.canvas.mouse.current_shape = "oval"
    self.canvas.mouse.draw_bind_mouse_events()

def create_triangle(self):
    self.canvas.mouse.current_shape = "triangle"
    self.canvas.mouse.draw_bind_mouse_events()

```

In `shape_editor.py` add import for the Left Frame class and modify the left frame creation to use the custom Left Frame class.

```

import customtkinter as ctk

from UI_Lib.canvas import Canvas
from UI_Lib.top_frame import TopFrame
from UI_Lib.left_frame import LeftFrame


class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")

        self.shape_list = []

        # Add widgets to the app here
        self.canvas = Canvas()
        self.top_frame = TopFrame(self, self.canvas)
        self.left_frame = LeftFrame(self, self.canvas)

```

Run the program and verify that it runs without error. If we remove the GUI mock-up comments, the total lines of code for `shape_editor.py` has been reduced to a manageable 41 lines of code. The Mouse Module has the most lines of code at 125.

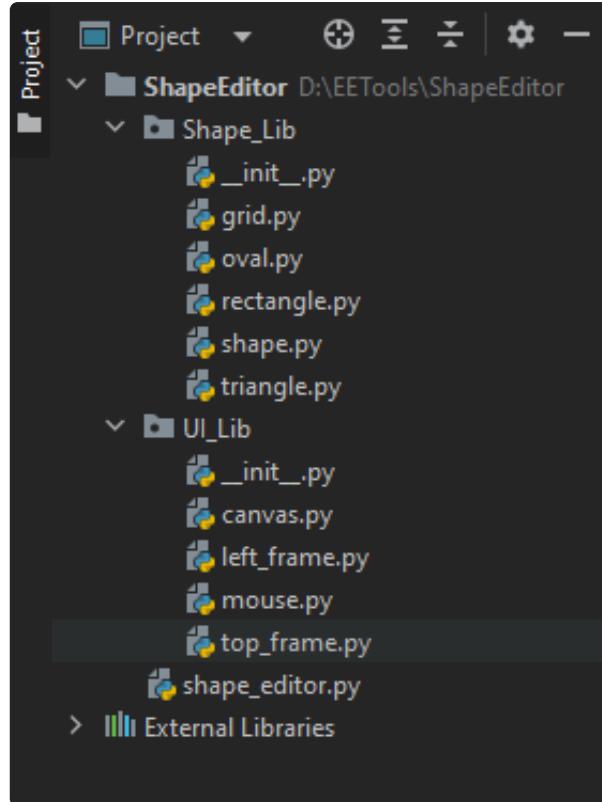
Summary

This concludes the Shape Editor project and all project features and specifications have been met. It is important to know when the project is complete. Feature creep can cause a

project to linger for much longer than the schedule allows. However, the application architecture is designed to add new features if needed in the future. The next project will add a line drawing capability to the Shape Editor as an example of adding a feature.

In case your code does not run or you got lost in the explanations above, here is the complete source code in modularized form.

Final directory structure



shape_editor.py

```
import customtkinter as ctk

from UI_Lib.canvas import Canvas
from UI_Lib.top_frame import TopFrame
from UI_Lib.left_frame import LeftFrame


class ShapeEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""

    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Shape Editor")
```

```

    self.shape_list = []

    # Add widgets to the app here
    self.canvas = Canvas()
    self.top_frame = TopFrame(self, self.canvas)
    self.left_frame = LeftFrame(self, self.canvas)

    self.top_frame.pack(side=ctk.TOP, fill="both", padx=5, pady=2)
    self.left_frame.pack(side=ctk.LEFT, fill="both", padx=5, pady=2)
    self.canvas.pack(fill="both", expand=True, padx=2, pady=2)

    # Mouse & keyboard bindings
    self.bind('<r>', self.canvas.rotate_shape_ccw)
    self.bind('<e>', self.canvas.rotate_shape_cw)
    self.bind('+', self.canvas.scale_up)
    self.bind('-', self.canvas.scale_down)
    self.bind("<Configure>", self.on_window_resize)

def on_window_resize(self, _event):
    self.canvas.draw_shapes()

if __name__ == "__main__":
    """Instantiate the Shape Editor application and run the main loop"""
    app = ShapeEditorApp()
    app.mainloop()

```

UI_Lib directory:

Empty `__init__.py` file

`canvas.py`

```

import customtkinter as ctk
from UI_Lib.mouse import Mouse
from Shape_Lib.grid import Grid

class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()

```

```

    self.shape_list = []
    self.mouse = Mouse(self)
    self.selected = None

    self.grid_size = 10
    self.grid = Grid(self, self.grid_size)
    self.grid.draw_grid()

def draw_shapes(self):
    self.delete('all')
    self.grid.draw_grid()
    for s in self.shape_list:
        s.draw()

def rotate_shape_ccw(self, _event):
    if self.selected is not None:
        self.selected.angle -= 90
        if self.selected.angle < 0:
            self.selected.angle = 270
    self.draw_shapes()

def rotate_shape_cw(self, _event):
    if self.selected is not None:
        self.selected.angle += 90
        if self.selected.angle > 270:
            self.selected.angle = 0
    self.draw_shapes()

def scale_up(self, _event):
    if self.selected and self.selected.scale < 10:
        self.selected.scale += 0.1
    self.draw_shapes()

def scale_down(self, _event):
    if self.selected and self.selected.scale > 0.1:
        self.selected.scale -= 0.1
    self.draw_shapes()

```

left_frame.py

```
import customtkinter as ctk
```

```

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self,
                                    text="Rectangle",
                                    command=self.create_rectangle)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

        oval_button = ctk.CTkButton(self,
                                    text="Oval",
                                    command=self.create_oval)
        oval_button.pack(side=ctk.TOP, padx=5, pady=5)

        tri_button = ctk.CTkButton(self,
                                   text="Triangle",
                                   command=self.create_triangle)
        tri_button.pack(side=ctk.TOP, padx=5, pady=5)

    def create_rectangle(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.canvas.mouse.draw_bind_mouse_events()

    def create_oval(self):
        self.canvas.mouse.current_shape = "oval"
        self.canvas.mouse.draw_bind_mouse_events()

    def create_triangle(self):
        self.canvas.mouse.current_shape = "triangle"
        self.canvas.mouse.draw_bind_mouse_events()

```

mouse.py

```

from Shape_Lib.rectangle import Rectangle
from Shape_Lib.oval import Oval
from Shape_Lib.triangle import Triangle

class Mouse:
    def __init__(self, canvas):

```

```
self.canvas = canvas

self.selected_obj = None
self.current_shape = None
self.current_shape_obj = None

self.start_x, self.start_y = 0, 0
self.offset_x1, self.offset_y1 = 0, 0
self.offset_x2, self.offset_y2 = 0, 0

def unbind_mouse_events(self):
    self.canvas.unbind("<Button-1>")
    self.canvas.unbind("<B1-Motion>")
    self.canvas.unbind("<ButtonRelease-1>")

def draw_bind_mouse_events(self):
    self.canvas.bind("<Button-1>", self.draw_left_down)
    self.canvas.bind("<B1-Motion>", self.draw_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

def bind_move_mouse_events(self):
    self.canvas.bind("<Button-1>", self.move_left_down)
    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

def move_left_down(self, event):
    x, y = event.x, event.y
    x, y = self.canvas.grid.snap_to_grid(x, y)
    self.select_shape(x, y)
    if self.canvas.selected:
        x1, y1 = self.canvas.selected.x1, self.canvas.selected.y1
        x2, y2 = self.canvas.selected.x2, self.canvas.selected.y2
        self.offset_x1 = event.x - x1
        self.offset_y1 = event.y - y1
        self.offset_x2 = event.x - x2
        self.offset_y2 = event.y - y2
        self.canvas.draw_shapes()

def move_left_drag(self, event):
    if self.canvas.selected:
        x = event.x - self.offset_x1
        y = event.y - self.offset_y1
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.canvas.selected.x1, self.canvas.selected.y1 = x, y
        x = event.x - self.offset_x2
        y = event.y - self.offset_y2
```

```

        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.canvas.selected.x2, self.canvas.selected.y2 = x, y
        self.canvas.draw_shapes()

def move_left_up(self, _event):
    self.offset_x1 = 0
    self.offset_y1 = 0
    self.offset_x2 = 0
    self.offset_y2 = 0

def draw_left_down(self, event):
    self.start_x = event.x
    self.start_y = event.y
    self.start_x, self.start_y = \
        (self.canvas.grid.snap_to_grid(self.start_x, self.start_y))

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas,
            self.start_x, self.start_y,
            self.start_x, self.start_y,
            fill_color="white",
            border_color="black",
            border_width=3)
    elif self.current_shape == "oval":
        self.current_shape_obj = Oval(self.canvas,
            self.start_x, self.start_y,
            self.start_x, self.start_y,
            fill_color="white",
            border_color="black",
            border_width=3)
    elif self.current_shape == "triangle":
        self.current_shape_obj = Triangle(self.canvas,
            self.start_x, self.start_y,
            self.start_x + 100, self.start_y + 100,
            fill_color="white",
            border_color="black",
            border_width=3)

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)
        self.canvas.draw_shapes()

def draw_left_drag(self, event):
    if self.current_shape_obj:
        x, y = event.x, event.y
        x, y = self.canvas.grid.snap_to_grid(x, y)

```

```
        self.current_shape_obj.x1, self.current_shape_obj.y1 =
self.start_x, self.start_y
        self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
        self.canvas.draw_shapes()

def draw_left_up(self, event):
    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.bind_move_mouse_events()

def select_shape(self, x, y):
    for s in self.canvas.shape_list:
        w = s.x2 - s.x1
        h = s.y2 - s.y1
        if (
            s.x1 <= x <= s.x2 + w
            and s.y1 <= y <= s.y2 + h
        ):
            self.unselect_all_objects()
            self.canvas.selected = s
            self.canvas.selected.is_selected = True

def unselect_all_objects(self):
    for s in self.canvas.shape_list:
        s.is_selected = False
    self.canvas.draw_shapes()
```

top_frame.py

```
                command=self.set_fill_color)
self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

self.border_color_button = ctk.CTkButton(self,
                                         text="Border Color",
                                         command=self.set_border_color)
self.border_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

border_width_optionmenu = ctk.CTkOptionMenu(self,
                                             values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                                             command=self.set_border_width)
border_width_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
border_width_optionmenu.set("3")

self.switch_var = ctk.StringVar(value="on")
switch = ctk.CTkSwitch(self, text="Grid Visible",
                        command=self.grid_visibility,
                        variable=self.switch_var,
                        onvalue="on", offvalue="off")
switch.pack(side=ctk.LEFT, padx=3, pady=3)

grid_size_optionmenu = ctk.CTkOptionMenu(self,
                                         values=[5, 10, 15, 20, 25, 40, 50,
                                                 60, 70, 80, 90, 100],
                                         command=self.set_grid_size)
grid_size_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
grid_size_optionmenu.set("10")

def set_fill_color(self):
    if self.canvas.selected:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.canvas.selected.fill_color = color
        self.canvas.draw_shapes()

def set_border_color(self):
    if self.canvas.selected:
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.canvas.selected.border_color = color
        self.canvas.draw_shapes()

def set_border_width(self, choice):
    if self.canvas.selected:
        self.canvas.selected.border_width = choice
        self.canvas.draw_shapes()
```

```
def grid_visibility(self):
    if self.switch_var.get() == "on":
        self.canvas.grid.grid_visible = True
    elif self.switch_var.get() == "off":
        self.canvas.grid.grid_visible = False
    self.canvas.draw_shapes()

def set_grid_size(self, choice):
    self.canvas.grid.grid_size = int(choice)
    self.canvas.draw_shapes()
```

Shape_Lib directory:

Empty `__init__.py` file

grid.py

```

# Creates all horizontal lines at intervals of 100
for i in range(0, h, self.grid_size):
    self.canvas.create_line([(0, i), (w, i)], dash=self.dash_list,
                          fill="#cccccc", tag='grid_line')

def snap_to_grid(self, x, y):
    if self.grid_visible:
        x = round(x / self.grid_size) * self.grid_size
        y = round(y / self.grid_size) * self.grid_size
    return x, y

```

oval.py

```

from Shape_Lib.shape import Shape

class Oval(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color,
                         border_width)

    def draw(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        w = w * self.scale
        h = h * self.scale
        center = self.x1 + w/2, self.y1 + h/2
        if self.angle == 90 or self.angle == 180:
            points = [center[0] - h/2, center[1] - w/2,
                      center[0] + h/2, center[1] + w/2]
        else:
            points = [center[0] - w/2, center[1] - h/2,
                      center[0] + w/2, center[1] + h/2]

        self.id = self.canvas.create_oval(points,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

        if self.is_selected:
            sel_points = [points[0]-5, points[1]-5,
                          points[2]+5, points[3]+5]

```

```
        self.canvas.create_rectangle(sel_points, fill=None,
                                    outline="red", width=2)
```

rectangle.py

```
from Shape_Lib.shape import Shape

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2, fill_color,
                         border_color, border_width)

    def draw(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        w = w * self.scale
        h = h * self.scale
        center = self.x1 + w/2, self.y1 + h/2
        if self.angle == 90 or self.angle == 180:
            points = [center[0] - h/2, center[1] - w/2,
                      center[0] + h/2, center[1] + w/2]
        else:
            points = [center[0] - w/2, center[1] - h/2,
                      center[0] + w/2, center[1] + h/2]

        self.id = self.canvas.create_rectangle(points,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)

        if self.is_selected:
            sel_points = [points[0]-5, points[1]-5,
                          points[2]+5, points[3]+5]
            self.canvas.create_rectangle(sel_points, fill=None,
                                         outline="red", width=2)
```

shape.py

```

class Shape:
    def __init__(self, canvas, x1, y1, x2, y2, fill_color, border_color,
                 border_width):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = fill_color
        self.border_color = border_color
        self.border_width = border_width

        self.id = None
        self.is_selected = False
        self.angle = 0
        self.scale = 1.0 # Default is 100% scale factor, range 0.1 to 10.0

```

triangle.py

```

from Shape_Lib.shape import Shape


class Triangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2,
                         fill_color, border_color, border_width)
        self.points = []

    def draw(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        w = w * self.scale
        h = h * self.scale
        center = self.x1 + w/2, self.y1 + h/2
        if self.angle == 0:
            x1, y1 = (center[0], center[1] - h/2)
            x2, y2 = (center[0] + w / 2, center[1] + h / 2)
            x3, y3 = (center[0] - w / 2, center[1] + h / 2)
        elif self.angle == 90:
            x1, y1 = (center[0] - w / 2, center[1])
            x2, y2 = (center[0] + w / 2, center[1] + h / 2)
            x3, y3 = (center[0] + w / 2, center[1] - h / 2)
        elif self.angle == 180:

```

```
x1, y1 = (center[0], center[1] + h/2)
x2, y2 = (center[0] + w / 2, center[1] - h / 2)
x3, y3 = (center[0] - w / 2, center[1] - h / 2)
elif self.angle == 270:
    x1, y1 = (center[0] + w / 2, center[1])
    x2, y2 = (center[0] - w / 2, center[1] - h / 2)
    x3, y3 = (center[0] - w / 2, center[1] + h / 2)
points = [x1, y1, x2, y2,x3, y3]
self.id = self.canvas.create_polygon(points,
                                      fill=self.fill_color,
                                      outline=self.border_color,
                                      width=self.border_width)

if self.is_selected:
    sel_points = [center[0] - w/2 - 5, center[1] - h/2 - 5,
                  center[0] + w/2 + 5, center[1] + h/2 + 5]
    self.canvas.create_rectangle(sel_points, fill=None,
                                 outline="red", width=2)
```

On to the Line Editor Project.

Chapter 4 - Line Editor Project

This second intermediate project is a line editor focused on designing and testing line drawing functionality. Line drawing is not as easy as it sounds as we will not only draw lines but "connect" those lines to shape "connectors". When a shape such as a rectangle is moved, rotated, or resized, the connected lines will automatically resize to stay visually connected to the shape. The line editor will implement straight lines, three segment lines, and elbow lines. Direction control will be provided for the segmented line and elbow line to select horizontal or vertical orientation when the line is drawn.

Project Design, Features, and Specifications

Approach:

- ✓ Start with new Line Editor App
- ✓ Create a canvas
- ✓ Create a top frame
 - ✓ Add controls to change shape fill and border color as well as border width
 - ✓ Add grid on/off and grid size controls
- ✓ Create a left frame

Rectangle Features:

- ✓ Select and move rectangle with left mouse button
- ✓ Unselect rectangle with left mouse button, if selected
- ✓ Rotate rectangle with 'r' key
- ✓ Resize rectangle with left mouse button
- ✓ Show selectors when selected that can be used to resize it
- ✓ Snap-to-grid for move and resize
- ✓ Create new rectangle by drawing rectangle with left mouse button

Line Features:

- ✓ Draw new line with left mouse button
- ✓ Options for straight or segmented line or elbow line

- Select line, show selectors
- Unselect line with left mouse button, if selected
- Move line with left mouse button
- Resize selected line ends with left mouse button
- Snap-to-Grid for move and resize
- Snap-to-Shape if begin or end point hits a shape connector
 - Move line end when connected shape is moved, resized, or rotates
 - If segmented or elbow line, change line direction using 'h' and 'v' keys

Mouse Features:

- Adjust draw bindings to draw rectangle and line

Project Setup

Create a new project.

- Open PyCharm and create a new project called LineEditor using Python 3.11.5 interpreter in the "venv"
- From a terminal in PyCharm, install CustomTkinter
 - pip install customtkinter
 - The installation may already be satisfied by the installation during the previous projects.

My Project Directory: D:/EETools/LineEditor

GUI Design and Layout

The basic layout for the Line Editor will be the same as the Shape Editor in Chapter 3. Our objective for this section are

- Create a CustomTkinter window
- Add a Canvas
- Add a Top Frame
- Add a Left Frame

Create a new file in the Line Editor project called `line_editor.py`. This will be the main program file for the project. Initially, we will import the CustomTkinter library and give it an alias called "ctk". Next create the Line Editor App class which is derived from the CTk window class. Set the window geometry, i.e. size and location, to 800 x 600 px with x, y location at 100, 100 on the canvas. We set the window title to "Line Editor" which is displayed in the window caption. Next, create the three canvas widgets: canvas, top frame, and left frame. We will use the pack geometry layout method to place the widgets on the window and ensure that they fill and expand if the window is resized. Note that the order of the pack statements is important. Finally, create the main entry point that creates the app object and starts the app main program loop.

line_editor.py

```
import customtkinter as ctk

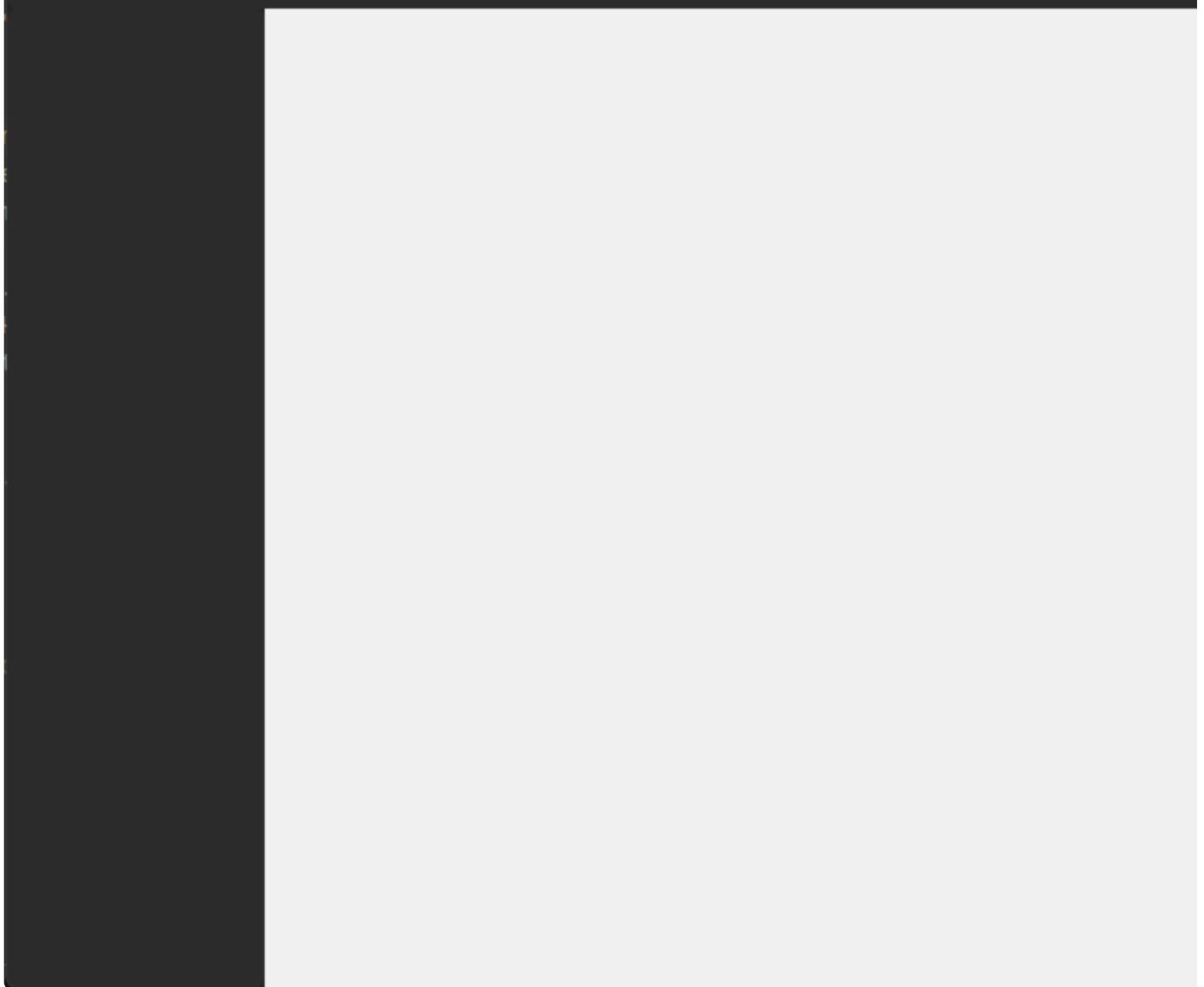
class LineEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main window"""
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Line Editor")

    # Create Canvas widget
    self.canvas = ctk.CTkCanvas(self)
    top_frame = ctk.CTkFrame(self, height=30)
    left_frame = ctk.CTkFrame(self, width=150)

    top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
    left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
    self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

if __name__ == "__main__":
    # Instantiate the Line Editor application and run the main loop
    app = LineEditorApp()
    app.mainloop()
```

We will create custom classes for the canvas, top frame, and left frame as we develop this project. Run the program which displays the main window.



Rectangle Class

Let's test the canvas by creating a Rectangle Class with a draw() method that draws itself on the canvas. Note that we could use the `canvas.create_rectangle()` method directly but we want to start implementing the object-oriented shape drawing framework described in the [Classes and Object-Oriented Programming \(OOP\)](#).

The objectives for this section are:

- Create a Rectangle class
- Draw a cyan rectangle on the canvas

Add a rectangle class to the Line Editor file above the definition for the Line Editor App class. Create a rectangle object in the Line Editor App class just below the code for creating the GUI widgets.

```
import customtkinter as ctk

class Rectangle:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)

class LineEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Line Editor")

        # Create Canvas widget
        self.canvas = ctk.CTkCanvas(self)
        top_frame = ctk.CTkFrame(self, height=30)
        left_frame = ctk.CTkFrame(self, width=150)

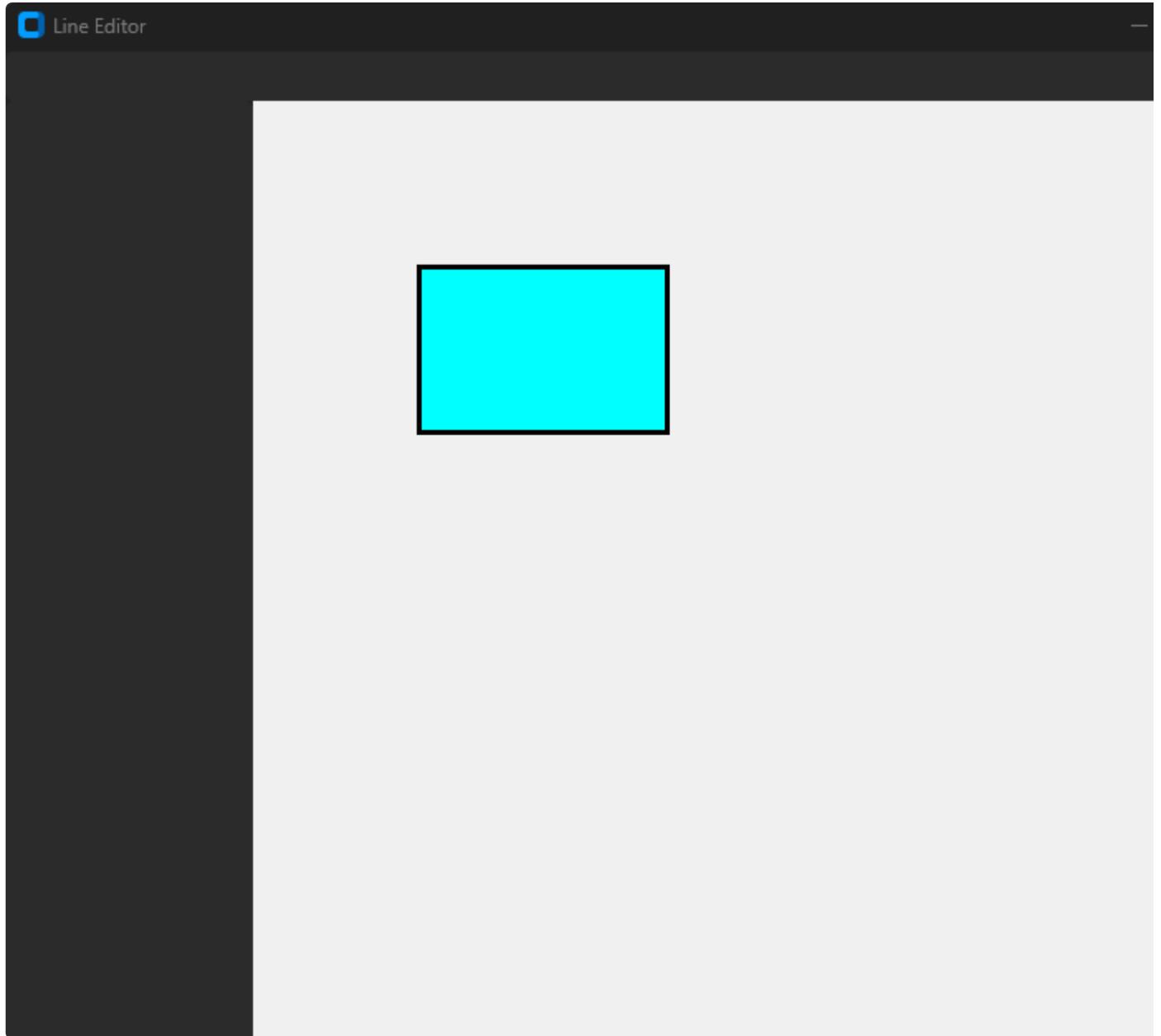
        top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Add shapes here
        rect = Rectangle(self.canvas, 100, 100, 250, 200)
        rect.draw()

if __name__ == "__main__":
```

```
# Instantiate the Line Editor application and run the main loop
app = LineEditorApp()
app.mainloop()
```

Run the program and verify that a cyan rectangle is draw on the canvas.



Line Class

Now that we have confirmed that we can draw shapes on the canvas, let's draw a line which is the main subject of this project.

The objectives for this section are:

- Create a Line class
- Draw a black line on the canvas

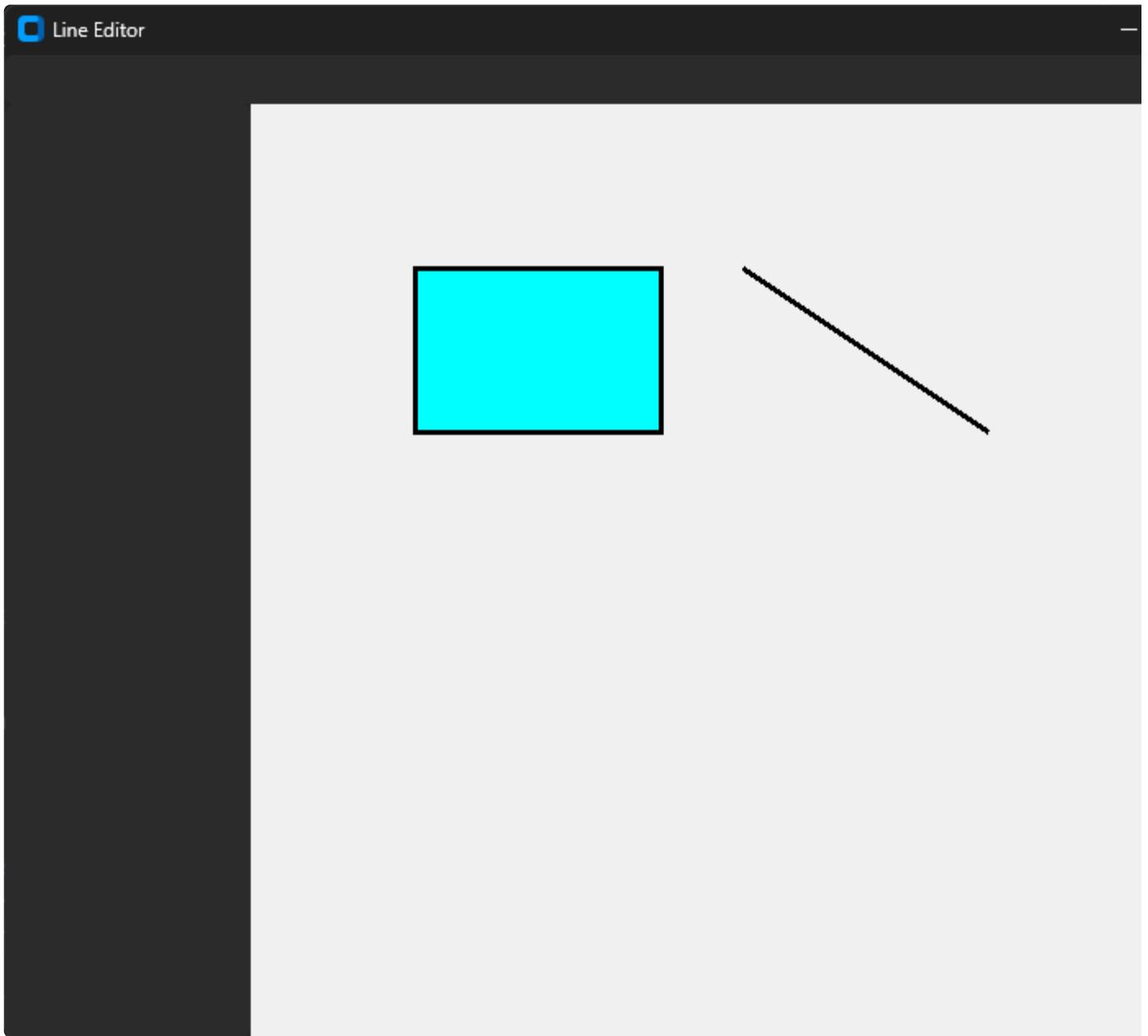
As we did with the rectangle class, add a line class to the Line Editor file above the definition for the Line Editor App class. Create a line object in the Line Editor App class just below the code for creating the rectangle.

Line Class

```
class Line:  
    def __init__(self, canvas, x1, y1, x2, y2):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.points = [self.x1, self.y1, self.x2, self.y2]  
  
    def draw(self):  
        self.points = [self.x1, self.y1, self.x2, self.y2]  
        self.canvas.create_line(self.points, fill="black", width=3)
```

Draw Line

```
# Add shapes here  
rect = Rectangle(self.canvas, 100, 100, 250, 200)  
rect.draw()  
  
line = Line(self.canvas, 300, 100, 450, 200)  
line.draw()
```



Great. Now that we can draw rectangles and lines on the screen, lets create a custom canvas class the will hold the shape list and provide a method to iterate over the shape list and call each shapes draw() method.

Canvas Class

The objectives for this class are:

- Create a custom Canvas class that inherits from `ctk.CTkCanvas`
- Add shapes to the canvas shape list
- Draw shapes on the canvas from the shape list

Canvas Class

```
class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()
        self.shape_list = []

    def draw_shapes(self):
        self.delete('all')
        for shape in self.shape_list:
            shape.draw()
```

Note that in the draw shapes method, we delete all shapes before redrawing them. We will use this method a lot during mouse draw, drag, and resize operations and we need to keep the canvas clean. Python can redraw shapes very quickly such that dynamic changes to shape size and position can be redrawn between updates. Sort of like a game loop in 2D and 3D games written in Python.

Next we need to modify the Line Editor App to create the canvas using our custom canvas class, add shapes to the canvas shape list, and call the draw shapes method.

```
# Create Canvas widget
self.canvas = Canvas() # Custom Canvas
top_frame = ctk.CTkFrame(self, height=30)
left_frame = ctk.CTkFrame(self, width=150)

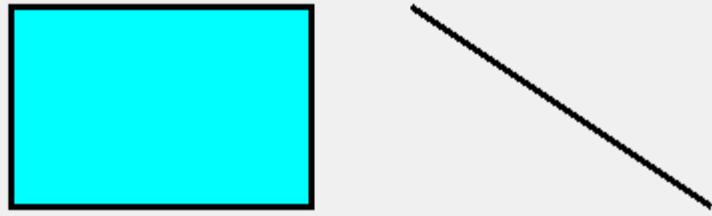
top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

# Add shapes here
rect = Rectangle(self.canvas, 100, 100, 250, 200)
self.canvas.shape_list.append(rect)

line = Line(self.canvas, 300, 100, 450, 200)
self.canvas.shape_list.append(line)

self.canvas.draw_shapes()
```

Run the program and confirm that the rectangle and line shapes are drawn on the canvas as before.



Point Class

A point represents an x, y coordinate on the canvas.

Points can be represented in many ways:

- Separate variables for x & y: `point_x`, `point_y`
- As a list or tuple
 - `point = list[x, y]`, access x coordinate: `point[0]`, access y coordinate: `point[1]`

However, our objective is to create a Point class that allows the programmer to use dot notation to access x & y coordinates:

Point Class

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

Test the Point Class in the Line Editor App

```
# Test point class  
a_point = Point(100, 100)  
print("Point: ", a_point.x, a_point.y)
```

Console output

```
Point: 100 100
```

Let's override the default print() result to format the output

Point Class

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __repr__(self):  
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Test the Point Class in the Line Editor App

```
# Test point class  
a_point = Point(100, 100)  
print("Point: ", a_point)
```

Console output

```
Point: (100, 100)
```

Nice! This will come in handy as we define various points in the program. For example, we can define the center point of a shape as follows:

```
center = Point(self.x1 + self.w/2, self.y1 +self.h/2)
```

Mouse Class

The mouse class will handle all of the mouse related operations to define the dynamic behavior of the Line Editor.

The objectives for this section are:

- Create a custom Mouse class
 - Create a select shape method in the Mouse class
 - Select shape on mouse left button down
 - Unselect shape, if already selected on mouse left button down
- Create a mouse object in the Canvas class
- Bind the mouse select events in the Line Editor App
- Rectangle and Line Classes - Draw selector box if shape is selected

The custom Mouse Class will need a reference to the canvas provided in the class initializer. We will keep track of which shape is selected in the `self.selected_obj` variable. We will use the approach of binding and unbinding mouse events based on the current operation. The first operation implemented is the "select" shape operation with mouse bindings defined in the `select_bind_mouse_events()` method which unbinds any existing mouse events and then binds the left mouse button (`<Button-1>`) to the `select_left_down()` event handler.

The `select_left_down()` method checks to see if a shape is already selected, if selected, the shape is unselected. If a shape is not selected, the select shape method is called which checks to see if the mouse x, y coordinates are within any shape boundary sort of like a hit test. If a shape is hit, the shape is selected.

```

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_obj = None

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def select_bind_mouse_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.select_left_down)

    def select_left_down(self, event):
        if self.selected_obj is not None:
            a_shape = self.selected_obj
            a_shape.is_selected = False
            self.selected_obj = None
            self.canvas.draw_shapes()
        else:
            x, y = event.x, event.y
            self.select_shape(x, y)
            self.canvas.draw_shapes()

    def select_shape(self, x, y):
        self.unselect_all_shapes()
        for shape in self.canvas.shape_list:
            if shape.x1 <= x <= shape.x2 and shape.y1 <= y <= shape.y2:
                shape.is_selected = True
                self.selected_obj = shape
                self.canvas.draw_shapes()

    def unselect_all_shapes(self):
        for shape in self.canvas.shape_list:
            shape.is_selected = False

        self.selected_obj = None
        self.canvas.draw_shapes()

```

To verify that a shape is selected, we will modify the Rectangle and Line classes to display a "selector". The selector in this case will be a red border rectangle with a transparent fill, i.e. `fill=None`.

Rectangle Class

```
class Rectangle:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.is_selected = False

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)

        if self.is_selected:
            self.points = [self.x1-5, self.y1-5, self.x2+5, self.y2+5]
            self.canvas.create_rectangle(self.points, fill=None, outline="red",
width=2)
```

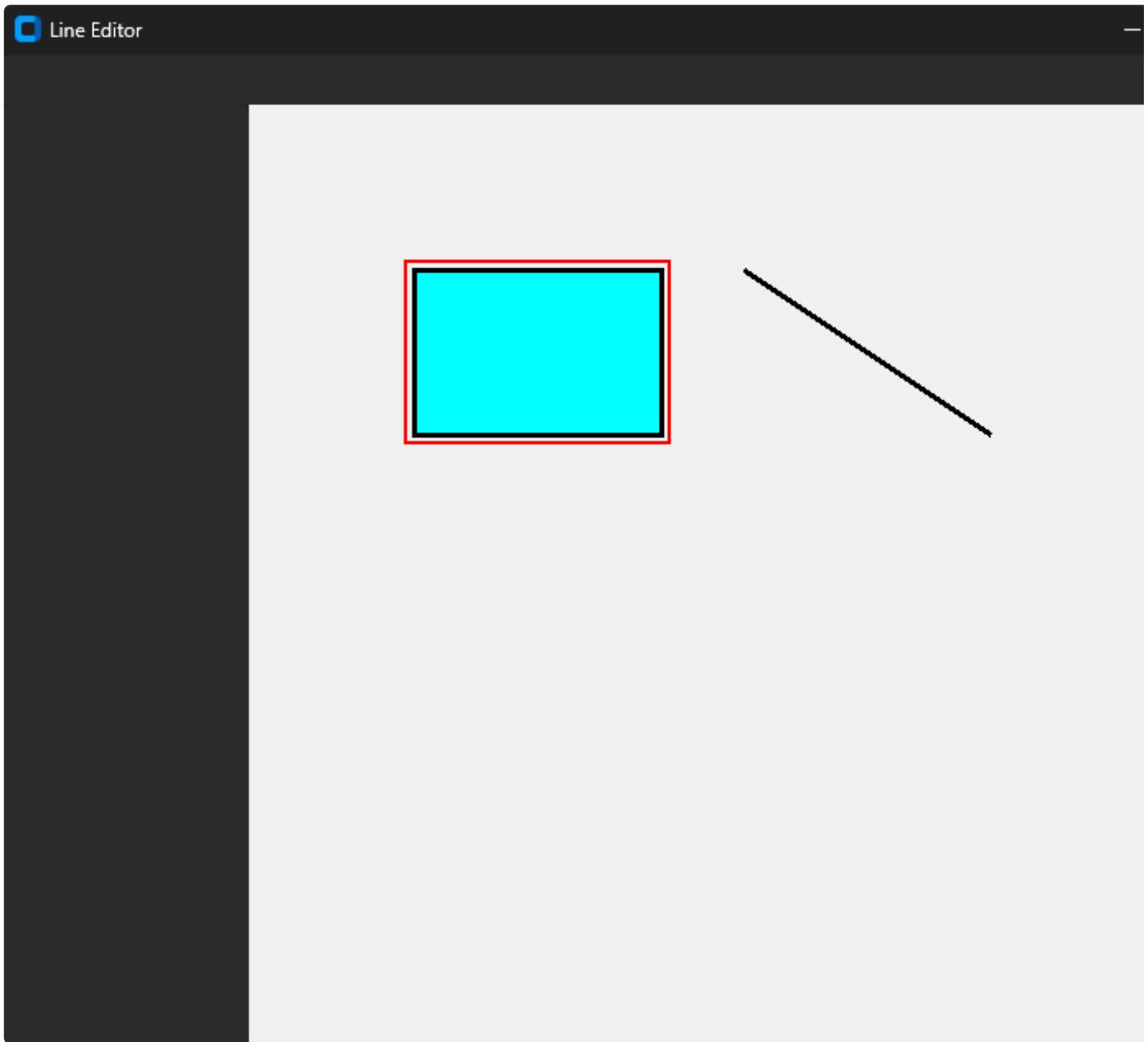
Line Class

```
class Line:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.is_selected = False

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_line(self.points, fill="black", width=3)

        if self.is_selected:
            self.points = [self.x1 - 5, self.y1 - 5, self.x2 + 5, self.y2 + 5]
            self.canvas.create_rectangle(self.points, fill=None, outline="red",
width=2)
```

A new variable called `self.is_selected` is added to both classes to control the visibility of the selector. The `is_selected` variable is set to True in the mouse select shape method.



Move Shape with Mouse

We will modify the Mouse Class to allow the user to select and move a shape with the left mouse button.

The objectives for this section are:

- Move a shape on the canvas with left mouse
 - Create a move left mouse down event

- Create a move left mouse drag event
- Create a move left mouse up event
- Bind move mouse events
- Only move the selected shape

The Mouse Class modifications include a `move_bind_mouse_events()` method that unbinds all existing bindings and adds three bindings for move left down, move left drag, and move left up. You will see this pattern again for implementation of shape resize operations.

The move left down method incorporates the shape selection code into the move handlers so the user can select and move a shape simultaneously. After checking to see if the shape was selected, it captures the mouse x, y position and calls the select shape method to check for a shape at that position. If a shape is found, offsets from the mouse position to the shape coordinates are calculate. All shapes are redrawn so that is a shape is selected or unselected, the selector is drawn or not drawn.

The move left drag method is where all the magic happens. If an object is selected, the offsets are used to calculate the shapes bounding coordinates: x1, y1, x2, y2. The shapes coordinates are updated and all shapes are redrawn which gives the effect that the shape is dragged on the canvas.

The move left up method fires when the user unclicks the left mouse button. It resets all offsets back to 0.

```
class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_obj = None

        self.start = Point(0,0)
        self.offset1 = Point(0,0)
        self.offset2 = Point(0,0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_bind_mouse_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
```

```

    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

def move_left_down(self, event):
    if self.selected_obj is not None:
        a_shape = self.selected_obj
        a_shape.is_selected = False
        self.selected_obj = None
        self.canvas.draw_shapes()

    x, y = event.x, event.y
    self.select_shape(x, y)
    if self.selected_obj:
        x1, y1 = self.selected_obj.x1, self.selected_obj.y1
        x2, y2 = self.selected_obj.x2, self.selected_obj.y2
        self.offset1.x = x - x1
        self.offset1.y = y - y1
        self.offset2.x = x - x2
        self.offset2.y = y - y2
        self.canvas.draw_shapes()

def move_left_drag(self, event):
    if self.selected_obj:
        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        self.selected_obj.x1, self.selected_obj.y1 = x, y
        x = event.x - self.offset2.x
        y = event.y - self.offset2.y
        self.selected_obj.x2, self.selected_obj.y2 = x, y
        self.canvas.draw_shapes()

def move_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0

def select_shape(self, x, y):
    self.unselect_all_shapes()
    for shape in self.canvas.shape_list:
        if shape.x1 <= x <= shape.x2 and shape.y1 <= y <= shape.y2:
            shape.is_selected = True
            self.selected_obj = shape
            self.canvas.draw_shapes()

def unselect_all_shapes(self):
    for shape in self.canvas.shape_list:
        shape.is_selected = False

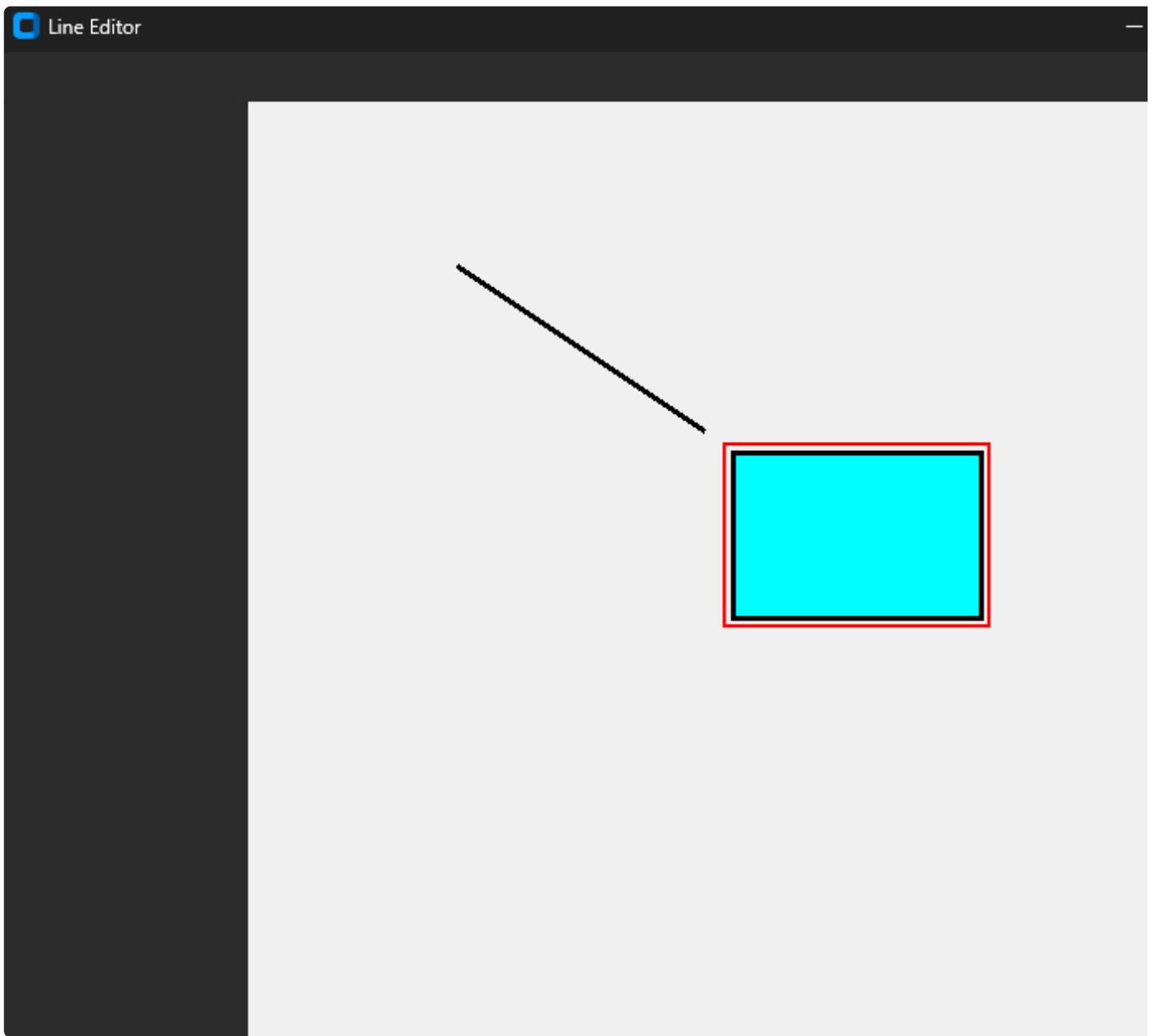
```

```
    self.selected_obj = None  
    self.canvas.draw_shapes()
```

In the Line Editor App Class call the move bind mouse events after the `canvas.draw_shapes` method.

```
class LineEditorApp(ctk.CTk):  
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main  
window"""  
    def __init__(self):  
        super().__init__()  
        self.geometry("800x600x100x100") # w, h, x, y  
        self.title("Line Editor")  
  
        # Create Canvas widget  
        self.canvas = Canvas()  
        top_frame = ctk.CTkFrame(self, height=30)  
        left_frame = ctk.CTkFrame(self, width=150)  
  
        top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)  
        left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)  
        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)  
  
        # Add shapes here  
        rect = Rectangle(self.canvas, 100, 100, 250, 200)  
        self.canvas.shape_list.append(rect)  
  
        line = Line(self.canvas, 300, 100, 450, 200)  
        self.canvas.shape_list.append(line)  
  
        self.canvas.draw_shapes()  
        self.canvas.mouse_move_bind_mouse_events() # Bind the mouse move  
events
```

Run the program and verify that you can move a shape with the left mouse button.



Great! Now we are getting somewhere. Lets see if we can rotate the rectangle.

Shape Rotation

Shapes such as rectangles, ovals, triangles, etc. can be rotated using the keyboard. For this project, we will rotate a shape in 90 deg increments. Lines will be rotated when we implement the resize operation.

The objectives of this section are:

- Bind 'r' key to rotate the shape ccw using a canvas method
- Add rotate method to shape

- Rotate by 90 degrees
- Update shape points list so selectors are drawn correctly

Add a rotate() method to the Rectangle Class. The idea here is to simply swap the shape height and width to simulate a 90 deg rotation. The selector will be drawn correctly because it is drawn using the updated shape coordinates.

```
class Rectangle:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.is_selected = False

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)

        if self.is_selected:
            self.points = [self.x1-5, self.y1-5, self.x2+5, self.y2+5]
            self.canvas.create_rectangle(self.points, fill=None, outline="red",
width=2)

    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - w/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2
```

We add a `rotate_shape()` method to the Canvas Class which is the handler when the 'r' key is pressed on the keyboard. It calls the shape rotate() method and redraws all shapes on the canvas.

```
class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()
        self.shape_list = []
        self.mouse = Mouse(self)
```

```

def draw_shapes(self):
    self.delete('all')
    for shape in self.shape_list:
        shape.draw()

def rotate_shape(self, _event):
    if not isinstance(self.mouse.selected_obj, Line):
        self.mouse.selected_obj.rotate()
    self.draw_shapes()

```

We add a binding for the 'r' key in the Line Editor App Class which calls the `canvas.rotate_shape()` method.

```

class LineEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""

    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Line Editor")

        # Create Canvas widget
        self.canvas = Canvas()
        top_frame = ctk.CTkFrame(self, height=30)
        left_frame = ctk.CTkFrame(self, width=150)

        top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

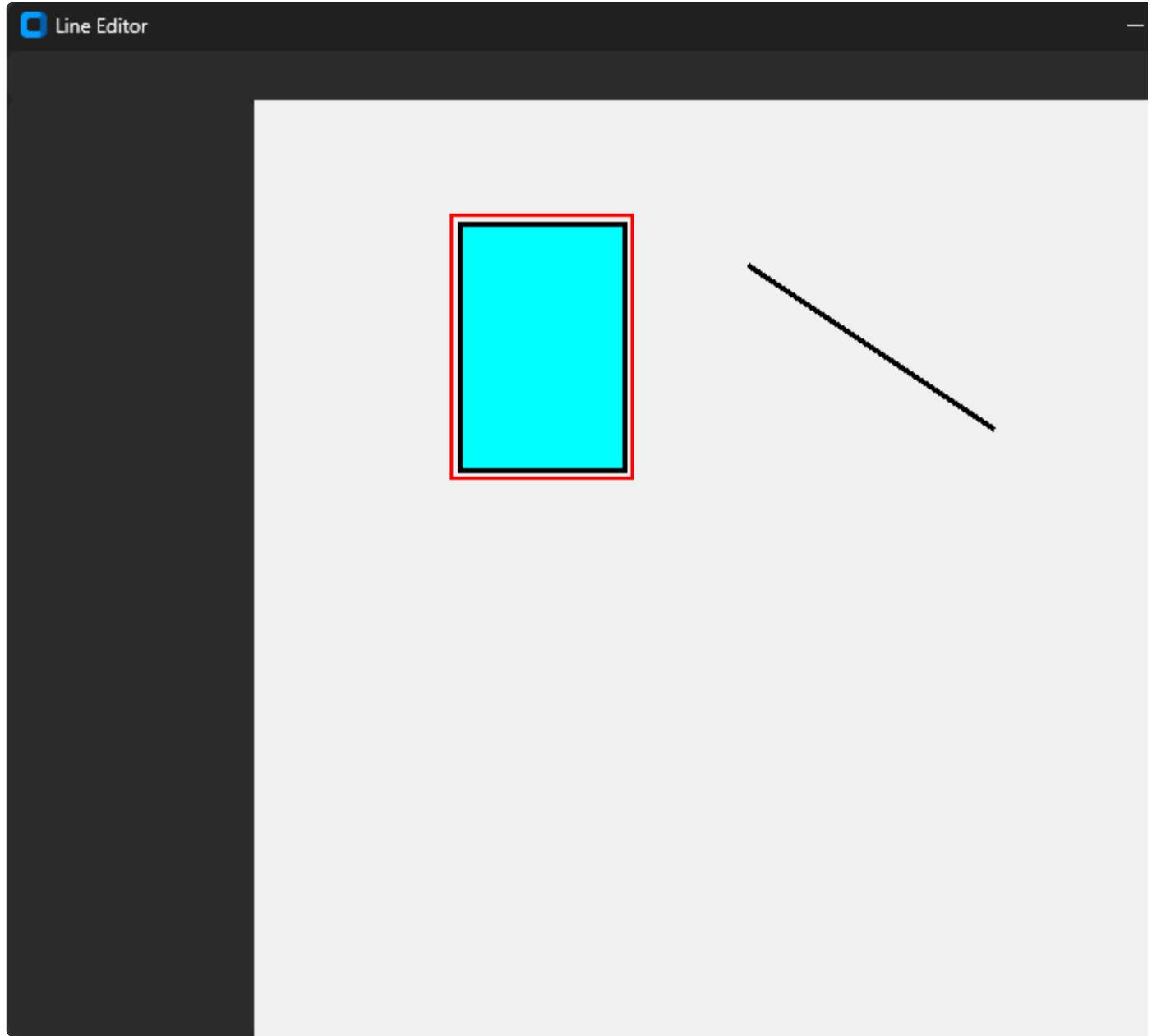
        # Add shapes here
        rect = Rectangle(self.canvas, 100, 100, 250, 200)
        self.canvas.shape_list.append(rect)

        line = Line(self.canvas, 300, 100, 450, 200)
        self.canvas.shape_list.append(line)

        self.canvas.draw_shapes()
        self.canvas.mouse.move_bind_mouse_events()
        self.bind('<r>', self.canvas.rotate_shape)

```

Run the program, select the rectangle, and press the 'r' key. Verify that the shape rotates by 90 deg and that the selector is also rotated.



Shape Base Class

Inspection of the Rectangle and Line classes indicates that duplicate code exists. We will create a Shape Class as a base class for shapes with common code. The Rectangle and Line classes will "inherit" the Shape Class code.

The objectives for this section are:

- Move common code from Rectangle and Line to a new Shape base class
- Modify Rectangle and Line to inherit from Shape

Create a new Shape Class and move common code from the Rectangle and Line classes to it. Essentially, all code from the derived class initializers are moved to the Shape Class initializer.

```
class Shape:  
    def __init__(self, canvas, x1, y1, x2, y2):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.points = [self.x1, self.y1, self.x2, self.y2]  
        self.is_selected = False
```

Modify the Rectangle Class to inherit from the Shape Class.

```
class Rectangle(Shape):  
    def __init__(self, canvas, x1, y1, x2, y2):  
        super().__init__(canvas, x1, y1, x2, y2)  
  
    def draw(self):  
        self.points = [self.x1, self.y1, self.x2, self.y2]  
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",  
width=3)  
  
        if self.is_selected:  
            self.points = [self.x1-5, self.y1-5, self.x2+5, self.y2+5]  
            self.canvas.create_rectangle(self.points, fill=None, outline="red",  
width=2)  
  
    def rotate(self):  
        w, h = self.x2 - self.x1, self.y2 - self.y1  
        center = Point(self.x1 + w / 2, self.y1 + h / 2)  
        self.x1, self.y1 = center.x - h/2, center.y - w/2  
        self.x2, self.y2 = center.x + h/2, center.y + w/2
```

Modify the Line Class to inherit from the Shape Class.

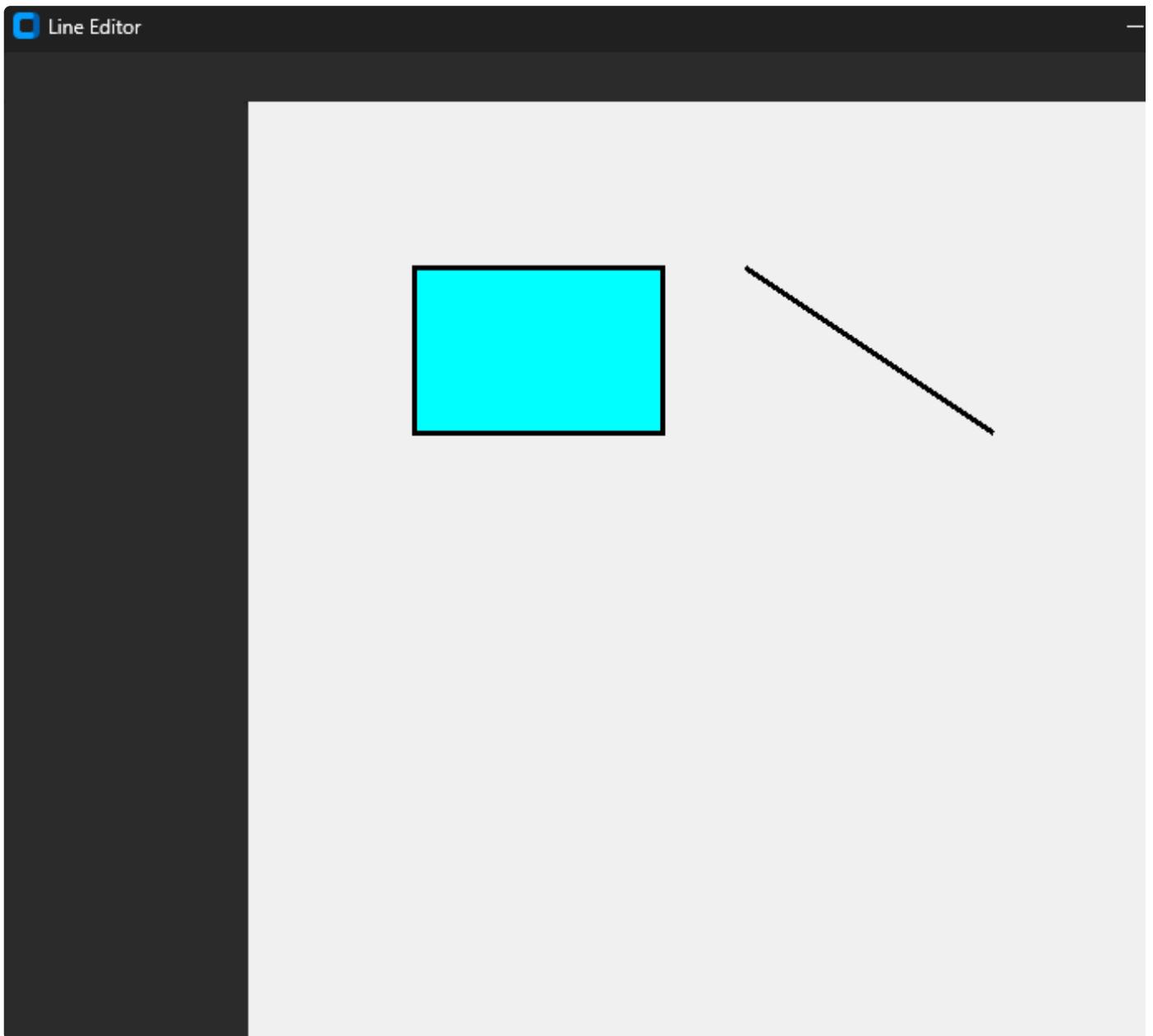
```
class Line(Shape):  
    def __init__(self, canvas, x1, y1, x2, y2):
```

```
super().__init__(canvas, x1, y1, x2, y2)

def draw(self):
    self.points = [self.x1, self.y1, self.x2, self.y2]
    self.canvas.create_line(self.points, fill="black", width=3)

    if self.is_selected:
        self.points = [self.x1 - 5, self.y1 - 5, self.x2 + 5, self.y2 + 5]
        self.canvas.create_rectangle(self.points, fill=None, outline="red",
width=2)
```

Run the program and confirm that the rectangle and line are drawn correctly.



Selector Class

In the next section, we will add a resize capability to Rectangle and Line classes. We need a way to select a point on the shape, drag it, and have the shape resize itself.

The objectives for this section are:

- In order to determine the resize selection point, we will create a new Selector class
- Each selector will have a `draw()` function that will draw a red oval
- The Selector class will have its own hit test called `selector_hit_test()`
- Each shape will have a `selector_list`
- Each shape will create selectors with the x, y coordinate at the desired selector position
- Each shape will have a `draw_selectors` method that will replace the current red rectangle selector

Create a new Selector Class that has a reference to the canvas, name, and x, y position coordinates. The selector is drawn as an oval with a radius of 5 px. A selector hit test method is added that checks to see if the mouse coordinates are within the selector.

```
class Selector:  
    def __init__(self, canvas, name, x, y):  
        self.canvas = canvas  
        self.name = name  
        self.x = x  
        self.y = y  
  
        self.radius = 5  
  
    def selector_hit_test(self, event_x, event_y):  
        x1, y1 = self.x - self.radius, self.y - self.radius  
        x2, y2 = self.x + self.radius, self.y + self.radius  
        if x1 <= event_x <= x2 and y1 <= event_y <= y2:  
            return True  
        else:  
            return False  
  
    def draw(self):  
        sel_points = [self.x - self.radius, self.y - self.radius, self.x +  
        self.radius, self.y + self.radius]  
        self.canvas.create_oval(sel_points, fill="red", outline="black",  
        width=2)
```

Add a selector list to the Shape Class called `self.sel_list`.

```
class Shape:  
    def __init__(self, canvas, x1, y1, x2, y2):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.points = [self.x1, self.y1, self.x2, self.y2]  
        self.is_selected = False  
  
        self.sel_list = []
```

In the Rectangle Class create two new methods called `create_selectors()` and `draw_selectors()`. The create selectors method will create four selectors positioned at the corners of the shape and add them to the selector list. The draw selectors method will update each selector position and then iterate over the selector list and call its `draw()` method.

```
class Rectangle(Shape):  
    def __init__(self, canvas, x1, y1, x2, y2):  
        super().__init__(canvas, x1, y1, x2, y2)  
  
        # Create 4 selectors  
        self.s1, self.s2, self.s3, self.s4 = None, None, None, None  
        self.create_selectors()  
  
    def draw(self):  
        self.points = [self.x1, self.y1, self.x2, self.y2]  
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",  
width=3)  
  
        if self.is_selected:  
            self.draw_selectors()  
  
    def create_selectors(self):  
        # Calculate position of selectors from current shape position  
        x1, y1, x2, y2 = self.points  
  
        # Create 4 selector objects: 4 corner of shape  
        self.s1 = Selector(self.canvas, "s1", x1, y1)  
        self.s2 = Selector(self.canvas, "s2", x2, y1)
```

```

        self.s3 = Selector(self.canvas, "s3", x2, y2)
        self.s4 = Selector(self.canvas, "s4", x1, y2)

        # Update the selector list
        self.sel_list = [self.s1, self.s2, self.s3, self.s4]

    def draw_selectors(self):
        # Recalculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Define 5 selectors: 4 shape corner - number of selectors is unique to
        # each shape
        self.s1.x, self.s1.y = x1, y1
        self.s2.x, self.s2.y = x2, y1
        self.s3.x, self.s3.y = x2, y2
        self.s4.x, self.s4.y = x1, y2

        # Draw the selectors
        for s in self.sel_list:
            s.draw()

    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - w/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

```

Similarly, in the Line Class create two new methods called `create_selectors()` and `draw_selectors()`. The create selectors method will create two selectors at the line ends and add them to the selector list. The draw selectors method will update each selector position and then iterate over the selector list and call its `draw()` method.

```

class Line(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_line(self.points, fill="black", width=3)

```

```
if self.is_selected:
    self.draw_selectors()

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 2 selector objects: 2 ends of the line
    self.s1 = Selector(self.canvas, "s1", x1, y1)
    self.s2 = Selector(self.canvas, "s2", x2, y2)

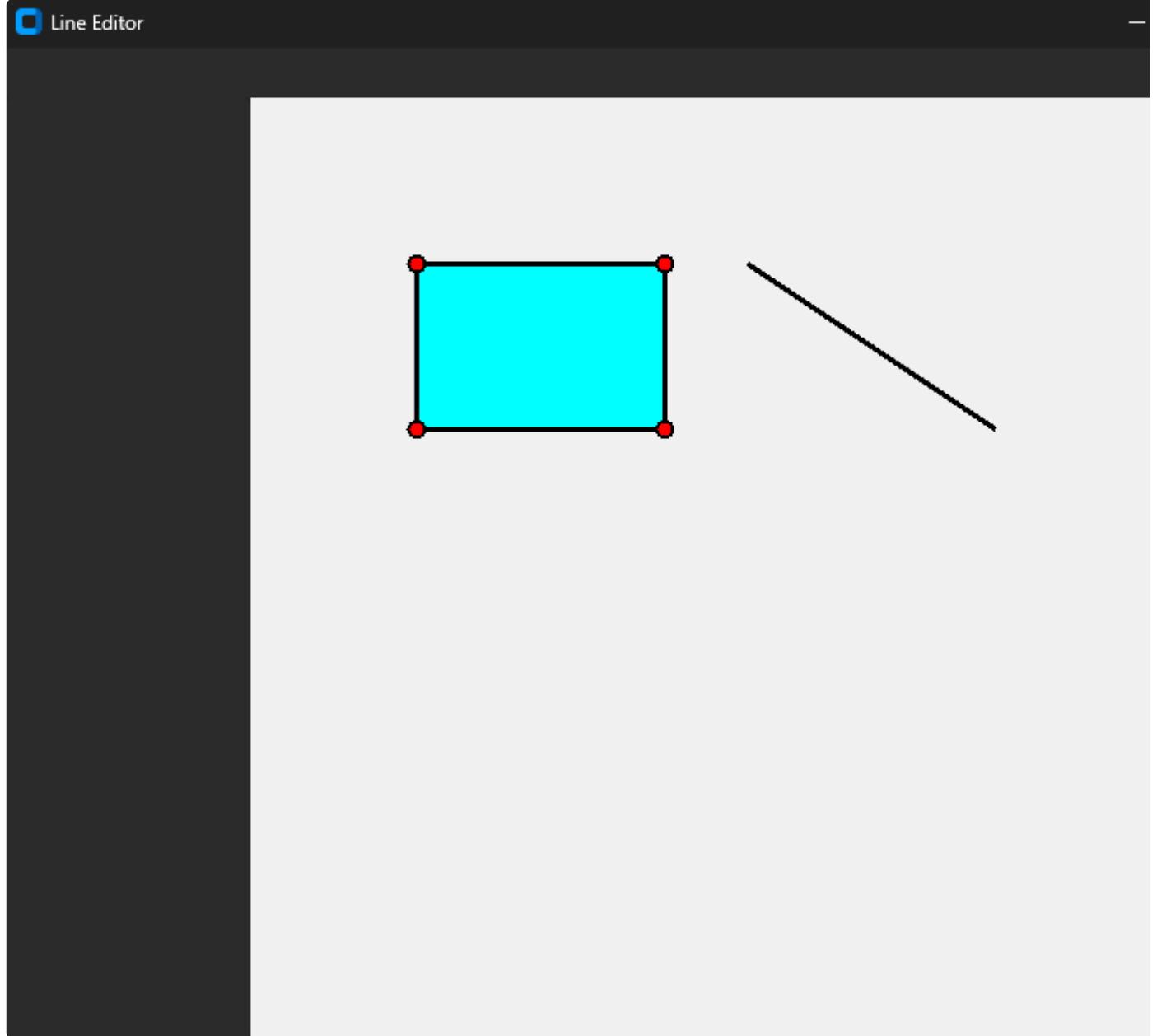
    # Update the selector list
    self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()
```

Run the program and confirm that the new selectors are drawn when either shape is selected and that the selectors are drawn correctly when the shape is moved.



Shape Resize

Now that we have shapes with selectors, we can implement the shape resize methods.

The objectives for this section are:

- Check for selector hit on selected shape
- If hit, bind mouse to resize mouse events
- Add resize method to Rectangle and Line classes

Add a check selector hit method to the Shape Class. This method will iterate over the selector list and call the selector hit test to see if a selector has been selected by the user.

If it has, the selector is returned to the calling program.

```
class Shape:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.is_selected = False

        self.sel_list = []
        self.id = None

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None
```

Add a resize() method to the Rectangle Class. This method checks to see if one of the four selectors has been selected by the user. If selected, the correct x, y coordinates are updated to the new position of the selector. This method is called by the resize event handlers in the Mouse Class.

```
class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 4 selectors
        self.s1, self.s2, self.s3, self.s4 = None, None, None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)

        if self.is_selected:
            self.draw_selectors()
```

```

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 4 selector objects: 4 corner of shape
    self.s1 = Selector(self.canvas, "s1", x1, y1)
    self.s2 = Selector(self.canvas, "s2", x2, y1)
    self.s3 = Selector(self.canvas, "s3", x2, y2)
    self.s4 = Selector(self.canvas, "s4", x1, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2, self.s3, self.s4]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 5 selectors: 4 shape corner - number of selectors is unique to
    each shape
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y1
    self.s3.x, self.s3.y = x2, y2
    self.s4.x, self.s4.y = x1, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2

```

```

        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        self.x1, self.y2 = x1, y2

```

Line Class

Add a `resize()` method to the Line Class. This method checks to see if one of the two selectors has been selected by the user. If selected, the correct x, y coordinates are updated to the new position of the selector. This method is called by the `resize` event handlers in the Mouse Class.

```

class Line(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_line(self.points, fill="black", width=3)

        if self.is_selected:
            self.draw_selectors()

    def create_selectors(self):
        # Calculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Create 2 selector objects: 2 ends of the line
        self.s1 = Selector(self.canvas, "begin", x1, y1)
        self.s2 = Selector(self.canvas, "end", x2, y2)

        # Update the selector list
        self.sel_list = [self.s1, self.s2]

    def draw_selectors(self):
        # Recalculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

```

```

# Define 2 selectors: 2 ends of the line
self.s1.x, self.s1.y = x1, y1
self.s2.x, self.s2.y = x2, y2

# Draw the selectors
for s in self.sel_list:
    s.draw()

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1

```

Modify the Mouse Class to add resize bind mouse events method that unbinds all current bindings and binds resize left down, resize left drag, and resize left up methods. The resize left down method calculates the offsets from the current mouse position to the shape coordinates. The resize left drag method creates an offset list that is passed as an argument to the shape resize() method which is called. All shapes are redrawn to simulate shape resize as the user holds the left mouse button and drags the mouse. The resize left up method resets all offsets to 0, unbinds the resize bindings, and sets the bindings to the move methods.

```

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_obj = None

        self.start = Point(0,0)
        self.offset1 = Point(0,0)
        self.offset2 = Point(0,0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_bind_mouse_events(self):

```

```

        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        if self.selected_obj:
            x, y = event.x, event.y
            sel = self.selected_obj.check_selector_hit(x, y)
            if sel:
                self.selected_obj.selector = sel.name
                self.unbind_mouse_events()
                self.bind_resize_mouse_events()
                self.resize_left_down(event)
                return
            else:
                a_shape = self.selected_obj
                a_shape.is_selected = False
                self.selected_obj = None
                self.canvas.draw_shapes()

            x, y = event.x, event.y
            self.select_shape(x, y)
            if self.selected_obj:
                x1, y1 = self.selected_obj.x1, self.selected_obj.y1
                x2, y2 = self.selected_obj.x2, self.selected_obj.y2
                self.offset1.x = x - x1
                self.offset1.y = y - y1
                self.offset2.x = x - x2
                self.offset2.y = y - y2
                self.canvas.draw_shapes()

    def move_left_drag(self, event):
        if self.selected_obj:
            x = event.x - self.offset1.x
            y = event.y - self.offset1.y
            self.selected_obj.x1, self.selected_obj.y1 = x, y
            x = event.x - self.offset2.x
            y = event.y - self.offset2.y
            self.selected_obj.x2, self.selected_obj.y2 = x, y
            self.canvas.draw_shapes()

    def move_left_up(self, _event):
        self.offset1.x, self.offset1.y = 0, 0
        self.offset2.x, self.offset2.y = 0, 0

```

```

def bind_resize_mouse_events(self):
    self.canvas.bind("<Button-1>", self.resize_left_down)
    self.canvas.bind("<B1-Motion>", self.resize_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

def resize_left_down(self, event):
    if self.selected_obj:
        x1, y1 = self.selected_obj.x1, self.selected_obj.y1
        x2, y2 = self.selected_obj.x2, self.selected_obj.y2
        self.offset1.x = event.x - x1
        self.offset1.y = event.y - y1
        self.offset2.x = event.x - x2
        self.offset2.y = event.y - y2
        self.canvas.draw_shapes()

def resize_left_drag(self, event):
    if self.selected_obj:
        offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
        self.offset2.y]
        self.selected_obj.resize(offsets, event)
        self.canvas.draw_shapes()

def resize_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0
    self.canvas.mouse.unbind_mouse_events()
    self.canvas.mouse.move_bind_mouse_events()

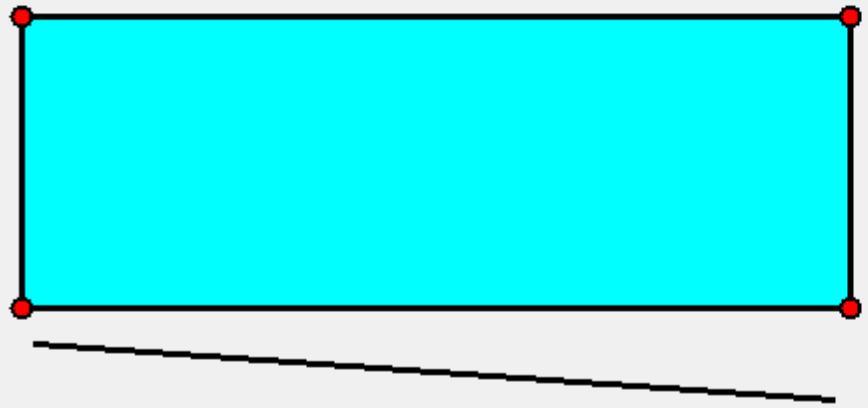
def select_shape(self, x, y):
    self.unselect_all_shapes()
    for shape in self.canvas.shape_list:
        if shape.x1 <= x <= shape.x2 and shape.y1 <= y <= shape.y2:
            shape.is_selected = True
            self.selected_obj = shape
            self.canvas.draw_shapes()

def unselect_all_shapes(self):
    for shape in self.canvas.shape_list:
        shape.is_selected = False

    self.selected_obj = None
    self.canvas.draw_shapes()

```

Run the program and confirm that the rectangle and line can be resized by selecting a selector and dragging the mouse.



Modularization

The `line_editor.py` file is now several hundred lines of code. This is a good time to "modularize" our code base.

The objectives for this section are:

- Create module directories: `UI_Lib`, `Shape_Lib`, `Helper_Lib`
- Move classes to module directories
- Add `__init__.py` to each module directory
- Add imports to each directory `__init__.py`
- Update files to import libraries from module directories

`line_editor.py` after shape and user-interface classes have been moved to modules.

```
import customtkinter as ctk
from UI_Lib import *
from Shape_Lib import *

class LineEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main window"""
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Line Editor")

        # Create Canvas widget
        self.canvas = Canvas()
        top_frame = ctk.CTkFrame(self, height=30)
        left_frame = ctk.CTkFrame(self, width=150)

        top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Add shapes here
        rect = Rectangle(self.canvas, 100, 100, 250, 200)
        self.canvas.shape_list.append(rect)

        a_line = Line(self.canvas, 300, 100, 450, 200)
        self.canvas.shape_list.append(a_line)

        self.canvas.draw_shapes()
        self.canvas.mouse_move_bind_mouse_events()
        self.bind('<r>', self.canvas.rotate_shape)

    if __name__ == "__main__":
        # Instantiate the Line Editor application and run the main loop
        app = LineEditorApp()
        app.mainloop()
```

Helper_Lib

Create a directory called Helper_Lib and add an `__init__.py` file which imports the Point Class.

`__init__.py`

```
# Import helper classes
from .point import Point
```

Create `point.py` and move the Point Class code to it.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Shape_Lib

Create a directory called Shape_Lib and add an `__init__.py` file which imports Shape, Selector, Rectangle, and Line Classes.

`__init__.py`

```
# import shape base class
from .shape import Shape

# import helper classes
from .selector import Selector

# import shape derived classes
from .rectangle import Rectangle

# import line classes
from .line import Line
```

Move the Shape Class code to the shape.py file.

```
class Shape:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.is_selected = False

        self.sel_list = []
        self.id = None

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None
```

Move the Rectangle Class code to rectangle.py file. Add import for the Shape Class, Selector Class, and Point Class.

```
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Helper_Lib.point import Point

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 4 selectors
        self.s1, self.s2, self.s3, self.s4 = None, None, None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)
```

```

    if self.is_selected:
        self.draw_selectors()

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 4 selector objects: 4 corner of shape
    self.s1 = Selector(self.canvas, "s1", x1, y1)
    self.s2 = Selector(self.canvas, "s2", x2, y1)
    self.s3 = Selector(self.canvas, "s3", x2, y2)
    self.s4 = Selector(self.canvas, "s4", x1, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2, self.s3, self.s4]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 5 selectors: 4 shape corner - number of selectors is unique to
    each shape
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y1
    self.s3.x, self.s3.y = x2, y2
    self.s4.x, self.s4.y = x1, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2

```

```

        y1 = event.y - offset_y1
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        self.x1, self.y2 = x1, y2

```

Move the Line Class code to the line.py file. Add imports for the Shape and Selector Classes.

```

from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector


class Line(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_line(self.points, fill="black", width=3)

        if self.is_selected:
            self.draw_selectors()

    def create_selectors(self):
        # Calculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Create 2 selector objects: 2 ends of the line
        self.s1 = Selector(self.canvas, "begin", x1, y1)
        self.s2 = Selector(self.canvas, "end", x2, y2)

        # Update the selector list

```

```

    self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1

```

Move Selector Class code to the selector.py file.

```

class Selector:
    def __init__(self, canvas, name, x, y):
        self.canvas = canvas
        self.name = name
        self.x = x
        self.y = y

        self.radius = 5

    def selector_hit_test(self, event_x, event_y):
        x1, y1 = self.x - self.radius, self.y - self.radius
        x2, y2 = self.x + self.radius, self.y + self.radius
        if x1 <= event_x <= x2 and y1 <= event_y <= y2:
            return True
        else:
            return False

```

```
def draw(self):
    sel_points = [self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius]
    self.canvas.create_oval(sel_points, fill="red", outline="black",
width=2)
```

UI_Lib

Create a directory called UI_Lib and add an `__init__.py` file which imports Canvas and Mouse Classes.

`__init__.py`

```
# Import user-interface libraries
from .canvas import Canvas
from .mouse import Mouse
```

Move the Canvas Class code to the canvas.py file. Add imports for CustomTkinter, Mouse, and Line Classes.

```
import customtkinter as ctk
from UI_Lib.mouse import Mouse
from Shape_Lib.line import Line

class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()
        self.shape_list = []
        self.mouse = Mouse(self)

    def draw_shapes(self):
        self.delete('all')
        for shape in self.shape_list:
            shape.draw()

    def rotate_shape(self, _event):
        if not isinstance(self.mouse.selected_obj, Line):
```

```
    self.mouse.selected_obj.rotate()
    self.draw_shapes()
```

Move the Mouse Class code to the mouse.py file. Add import for the Point Class.

```
from Helper_Lib.point import Point

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_obj = None

        self.start = Point(0,0)
        self.offset1 = Point(0,0)
        self.offset2 = Point(0,0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_bind_mouse_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        if self.selected_obj:
            x, y = event.x, event.y
            sel = self.selected_obj.check_selector_hit(x, y)
            if sel:
                self.selected_obj.selector = sel.name
                self.unbind_mouse_events()
                self.bind_resize_mouse_events()
                self.resize_left_down(event)
                return
            else:
                a_shape = self.selected_obj
                a_shape.is_selected = False
                self.selected_obj = None
                self.canvas.draw_shapes()
```

```

        x, y = event.x, event.y
        self.select_shape(x, y)
        if self.selected_obj:
            x1, y1 = self.selected_obj.x1, self.selected_obj.y1
            x2, y2 = self.selected_obj.x2, self.selected_obj.y2
            self.offset1.x = x - x1
            self.offset1.y = y - y1
            self.offset2.x = x - x2
            self.offset2.y = y - y2
            self.canvas.draw_shapes()

def move_left_drag(self, event):
    if self.selected_obj:
        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        self.selected_obj.x1, self.selected_obj.y1 = x, y
        x = event.x - self.offset2.x
        y = event.y - self.offset2.y
        self.selected_obj.x2, self.selected_obj.y2 = x, y
        self.canvas.draw_shapes()

def move_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0

def bind_resize_mouse_events(self):
    self.canvas.bind("<Button-1>", self.resize_left_down)
    self.canvas.bind("<B1-Motion>", self.resize_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

def resize_left_down(self, event):
    if self.selected_obj:
        x1, y1 = self.selected_obj.x1, self.selected_obj.y1
        x2, y2 = self.selected_obj.x2, self.selected_obj.y2
        self.offset1.x = event.x - x1
        self.offset1.y = event.y - y1
        self.offset2.x = event.x - x2
        self.offset2.y = event.y - y2
        self.canvas.draw_shapes()

def resize_left_drag(self, event):
    if self.selected_obj:
        offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
                  self.offset2.y]
        self.selected_obj.resize(offsets, event)

```

```
    self.canvas.draw_shapes()

def resize_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0
    self.canvas.mouse.unbind_mouse_events()
    self.canvas.mouse.move_bind_mouse_events()

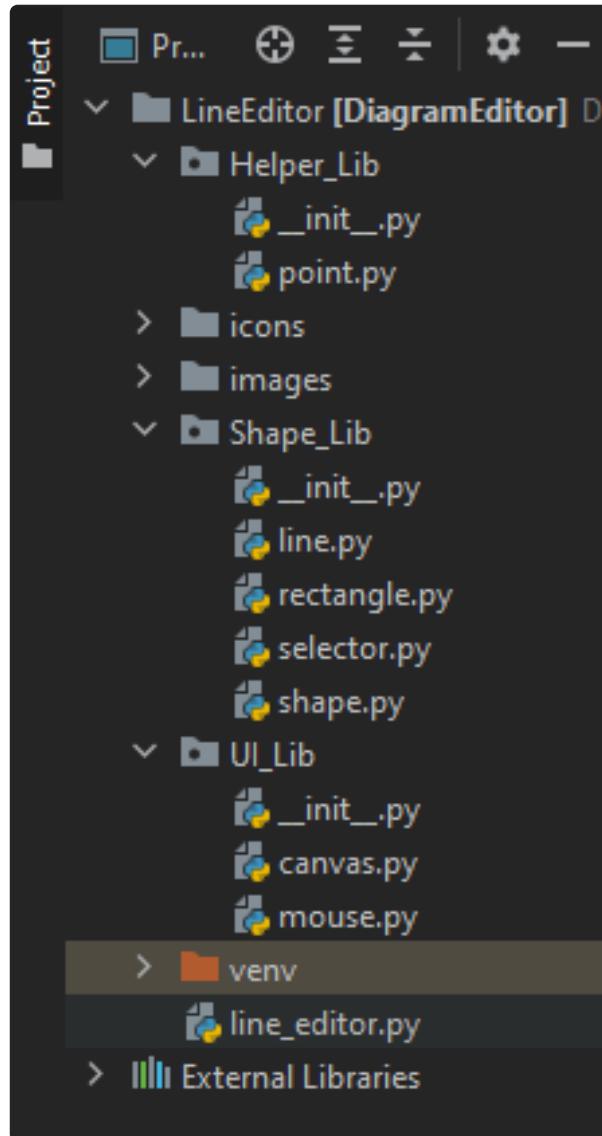
def select_shape(self, x, y):
    self.unselect_all_shapes()
    for shape in self.canvas.shape_list:
        if shape.x1 <= x <= shape.x2 and shape.y1 <= y <= shape.y2:
            shape.is_selected = True
            self.selected_obj = shape
            self.canvas.draw_shapes()

def unselect_all_shapes(self):
    for shape in self.canvas.shape_list:
        shape.is_selected = False

    self.selected_obj = None
    self.canvas.draw_shapes()
```

Mouse Class lines of code: 106 lines

Your project directory after modularizing the code base should look like this:



You can ignore the icons and images directories for now.

Left Frame Class

Shape creation is currently hard-coded in the Line Editor App Class. We want to give the user the ability to create shapes from a menu so that each diagram is unique. The Left Frame Class is a custom class derived from a `CTkFrame` widget which will hold the shape creation buttons. We will add a nice feature that will allow new shapes to be drawn on the canvas using a mouse.

The objectives for this section are:

- Create a new Left Frame Class

- Add buttons to create Rectangles and Lines
- Add draw mouse bindings to Mouse Class
- Remove hard-coded shapes from Line Editor App Class

Modify the `line_editor.py` code to use the custom Left Frame Class instead of the `ctk.CTkFrame` class.

```
import customtkinter as ctk
from UI_Lib import *

class LineEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main window"""
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Line Editor")

        # Create Canvas widget
        self.canvas = Canvas()
        top_frame = ctk.CTkFrame(self, height=30)
        a_left_frame = LeftFrame(self, self.canvas) # Using custom Left Frame
        Class

        top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        a_left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Declare mouse bindings
        self.bind('<r>', self.canvas.rotate_shape)

    if __name__ == "__main__":
        # Instantiate the Line Editor application and run the main loop
        app = LineEditorApp()
        app.mainloop()
```

Create a new Left Frame Class file called `left_frame.py`. The `LeftFrame` class is derived from the `ctk.CTkFrame` class. It is initialized with a parent and the canvas. Note that the parent is passed to the base class.

```

import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self,
                                    text="Rectangle",
                                    command=self.create_rectangle)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

        line_button = ctk.CTkButton(self,
                                    text="Line",
                                    command=self.create_line)
        line_button.pack(side=ctk.TOP, padx=5, pady=5)

    def create_rectangle(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.canvas.mouse.draw_bind_mouse_events()

    def create_line(self):
        self.canvas.mouse.current_shape = "line"
        self.canvas.mouse.draw_bind_mouse_events()

```

Add a `draw_bind_mouse_events()` method to the Mouse Class which binds three draw methods for new shape creation: `draw_left_down`, `draw_left_drag`, and `draw_left_up`. Add the `draw_left_down()` method that captures the start location for the new shape, creates the new shape based on the `current_shape` variable set by the Left Frame Class, and adds it to the canvas shape list. The `draw_left_drag()` method allows the user to resize the newly created shape by dragging the mouse. The `draw_left_up` method resets the current shape variables and rebinds the mouse events to the move mouse events.

```

from Helper_Lib.point import Point
from Shape_Lib.rectangle import Rectangle
from Shape_Lib.line import Line

class Mouse:

```

```

def __init__(self, canvas):
    self.canvas = canvas
    self.selected_obj = None
    self.current_shape = None
    self.current_shape_obj = None

    self.start = Point(0,0)
    self.offset1 = Point(0,0)
    self.offset2 = Point(0,0)

def unbind_mouse_events(self):
    self.canvas.unbind("<Button-1>")
    self.canvas.unbind("<B1-Motion>")
    self.canvas.unbind("<ButtonRelease-1>")

def draw_bind_mouse_events(self):
    self.canvas.bind("<Button-1>", self.draw_left_down)
    self.canvas.bind("<B1-Motion>", self.draw_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

def move_bind_mouse_events(self):
    self.unbind_mouse_events()
    self.canvas.bind("<Button-1>", self.move_left_down)
    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

def draw_left_down(self, event):
    self.unselect_all_shapes()
    self.start.x = event.x
    self.start.y = event.y

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
    self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "line":
        self.current_shape_obj = Line(self.canvas, self.start.x,
    self.start.y, self.start.x, self.start.y)

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)
        self.canvas.draw_shapes()

def draw_left_drag(self, event):
    if self.current_shape_obj:
        x, y = event.x, event.y
        self.current_shape_obj.x1, self.current_shape_obj.y1 =

```

```
self.start.x, self.start.y
    self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
    self.canvas.draw_shapes()

def draw_left_up(self, _event):
    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.move_bind_mouse_events()

def move_left_down(self, event):
    if self.selected_obj:
        x, y = event.x, event.y
        sel = self.selected_obj.check_selector_hit(x, y)
        if sel:
            self.selected_obj.selector = sel.name
            self.unbind_mouse_events()
            self.bind_resize_mouse_events()
            self.resize_left_down(event)
            return
        else:
            a_shape = self.selected_obj
            a_shape.is_selected = False
            self.selected_obj = None
            self.canvas.draw_shapes()

    x, y = event.x, event.y
    self.select_shape(x, y)
    if self.selected_obj:
        x1, y1 = self.selected_obj.x1, self.selected_obj.y1
        x2, y2 = self.selected_obj.x2, self.selected_obj.y2
        self.offset1.x = x - x1
        self.offset1.y = y - y1
        self.offset2.x = x - x2
        self.offset2.y = y - y2
        self.canvas.draw_shapes()

def move_left_drag(self, event):
    if self.selected_obj:
        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        self.selected_obj.x1, self.selected_obj.y1 = x, y
        x = event.x - self.offset2.x
        y = event.y - self.offset2.y
        self.selected_obj.x2, self.selected_obj.y2 = x, y
        self.canvas.draw_shapes()
```

```

def move_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0,0

def bind_resize_mouse_events(self):
    self.canvas.bind("<Button-1>", self.resize_left_down)
    self.canvas.bind("<B1-Motion>", self.resize_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

def resize_left_down(self, event):
    if self.selected_obj:
        x1, y1 = self.selected_obj.x1, self.selected_obj.y1
        x2, y2 = self.selected_obj.x2, self.selected_obj.y2
        self.offset1.x = event.x - x1
        self.offset1.y = event.y - y1
        self.offset2.x = event.x - x2
        self.offset2.y = event.y - y2
        self.canvas.draw_shapes()

def resize_left_drag(self, event):
    if self.selected_obj:
        offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
self.offset2.y]
        self.selected_obj.resize(offsets, event)
        self.canvas.draw_shapes()

def resize_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0
    self.canvas.mouse.unbind_mouse_events()
    self.canvas.mouse.move_bind_mouse_events()

def select_shape(self, x, y):
    self.unselect_all_shapes()
    for shape in self.canvas.shape_list:
        if shape.x1 <= x <= shape.x2 and shape.y1 <= y <= shape.y2:
            shape.is_selected = True
            self.selected_obj = shape
            self.canvas.draw_shapes()

def unselect_all_shapes(self):
    for shape in self.canvas.shape_list:
        shape.is_selected = False

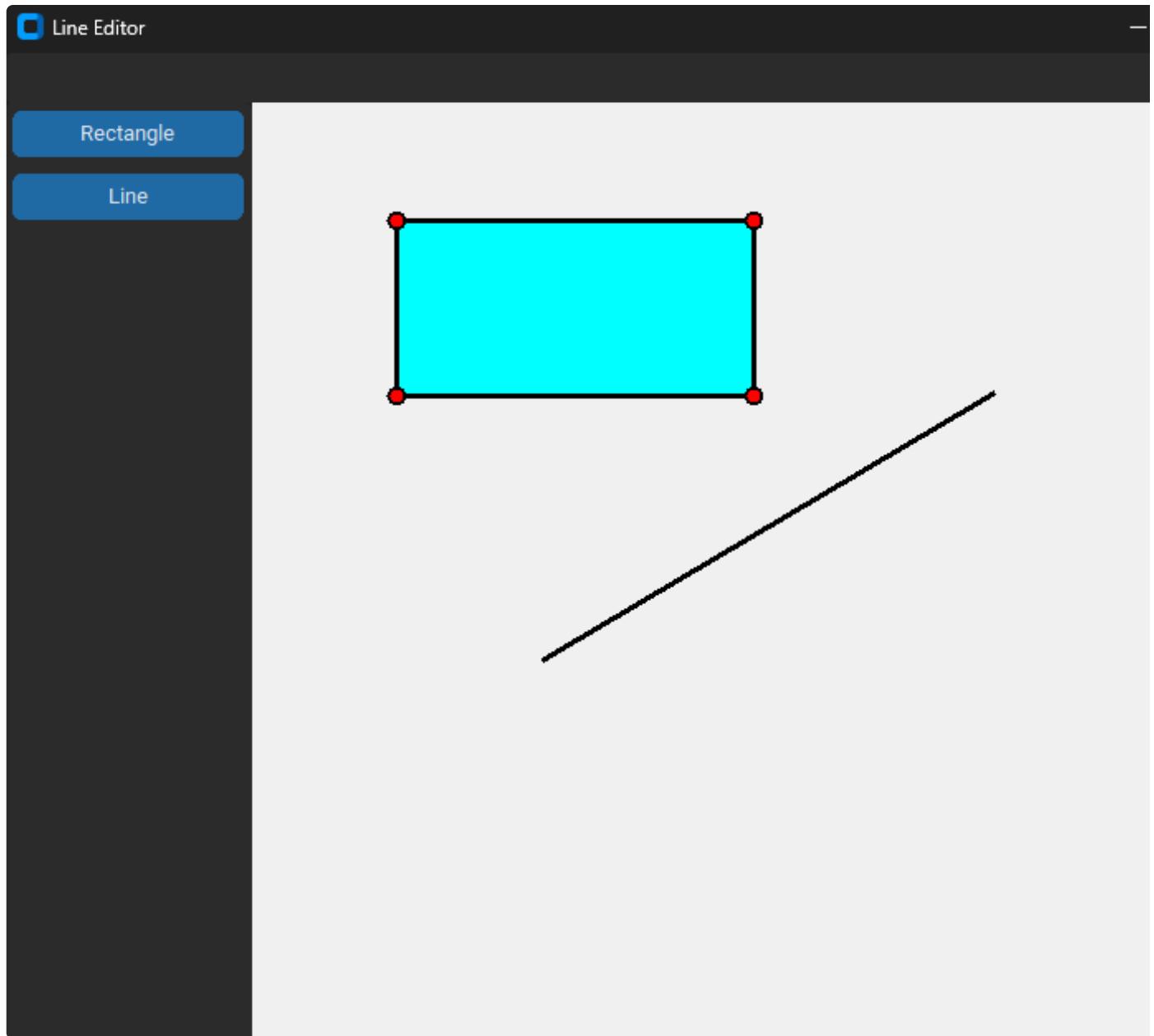
    self.selected_obj = None

```

```
self.canvas.draw_shapes()
```

Mouse Class lines of code = 141 lines

Run the program and verify that you can create a rectangle and a line using left menu.



Grid Class

We want to add a background "grid" to the application so that the user can precisely draw shape on the grid.

The objectives for this section are:

- Create a new Grid Class
 - Draw a "background" grid using either solid or dotted lines
 - Add grid draw() function to canvas draw_shapes()
 - Add binding in Line Editor App to redraw shapes if the window is configured
 - Add snap to grid method in the Grid Class
 - Create the grid object in the Canvas Class initializer

Create a new Grid Class in a file named grid.py. The Grid Class has a class global variable called type that will be used to determine if the grid is drawn with solid or dotted lines. The class initializer captures a reference to the canvas, sets the initial grid size, sets the default visibility to True, and define a `dash_list` variable which is modified based on the type variable. Add a `draw()` method that draws the grid lines from 0 to the canvas width in grid size steps and from 0 to the canvas height in grid size steps. Add the `snap_to_grid()` method which will adjust the provide x, y coordinates to the nearest grid x, y coordinates. This method will be used in the mouse draw, move, and resize methods to ensure that the shape is aligned to the grid if the grid is visible.

```

# Creates all horizontal lines at intervals of 100
for i in range(0, h, self.grid_size):
    self.canvas.create_line([(0, i), (w, i)], dash=self.dash_list,
                          fill="#cccccc", tag='grid_line')

def snap_to_grid(self, x, y):
    if self.grid_visible:
        x = round(x / self.grid_size) * self.grid_size
        y = round(y / self.grid_size) * self.grid_size
    return x, y

```

Add import for the Grid Class. Create a grid object in the Canvas Class initializer. Set the initial grid size to 10 px and draw the grid. Add a statement to the draw_shapes() method that draws the grid first before other shapes are drawn. This is how we ensure that the grid is drawn in the background behind all other shapes.

```

import customtkinter as ctk
from UI_Lib.mouse import Mouse
from Shape_Lib.line import Line
from Shape_Lib.grid import Grid

class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()
        self.shape_list = []
        self.mouse = Mouse(self)
        self.grid_size = 10
        self.grid = Grid(self, self.grid_size)
        self.grid.draw()

    def draw_shapes(self):
        self.delete('all')
        self.grid.draw()
        for shape in self.shape_list:
            shape.draw()

    def rotate_shape(self, _event):
        if not isinstance(self.mouse.selected_obj, Line):
            self.mouse.selected_obj.rotate()

```

```
    self.draw_shapes()
```

Modify the code in `line_editor.py` to bind the window resize event to the `on_window_resize()` handler. The handler calls the canvas draw shapes method whenever the size of the window is changed. This ensures that the grid is drawn to the edges of the window.

```
import customtkinter as ctk
from UI_Lib import *

class LineEditorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main window"""
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Line Editor")

        # Create Canvas widget
        self.canvas = Canvas()
        top_frame = ctk.CTkFrame(self, height=30)
        a_left_frame = LeftFrame(self, self.canvas)

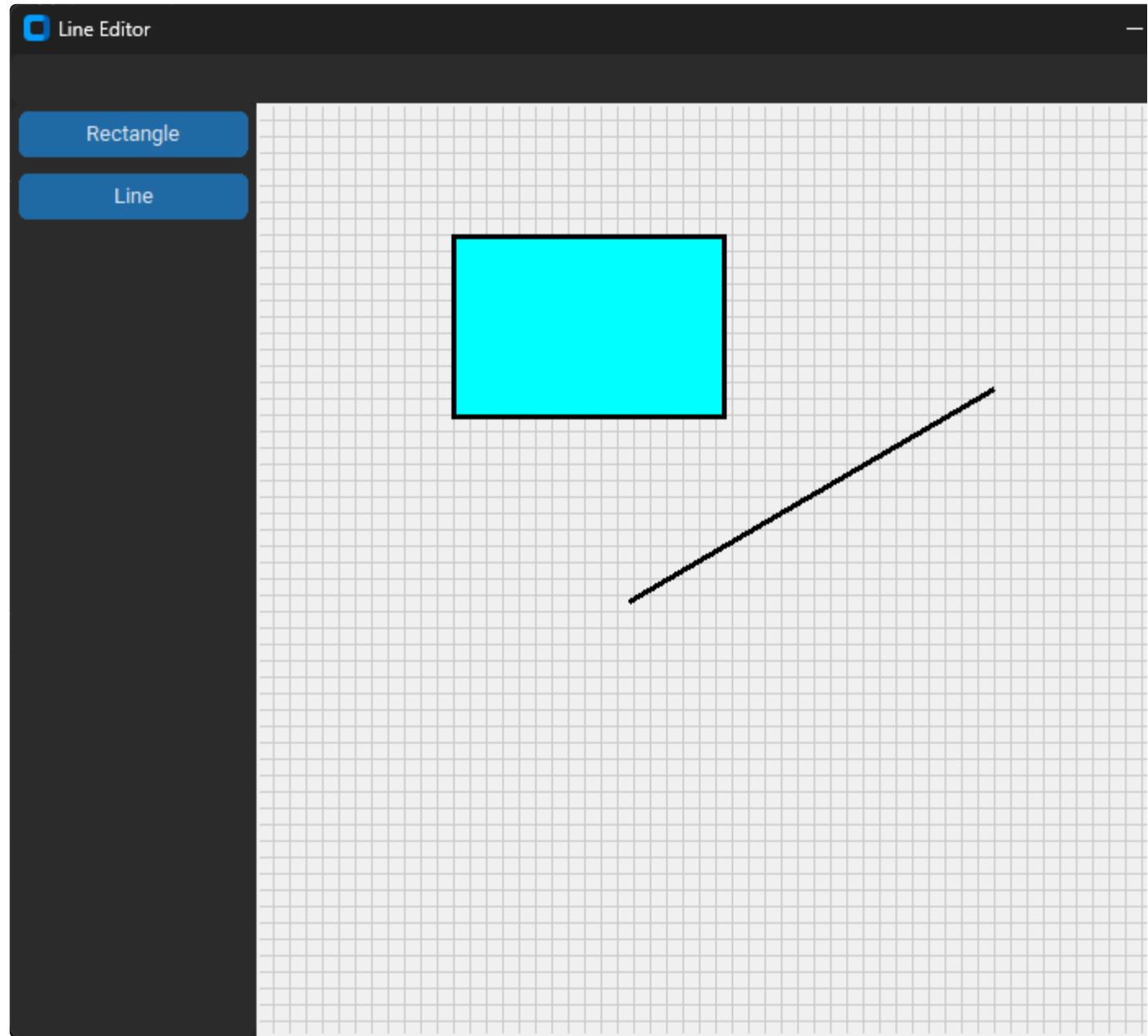
        top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        a_left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Declare mouse bindings
        self.bind('<r>', self.canvas.rotate_shape)
        self.bind("<Configure>", self.on_window_resize) # Bind to a windows
        resize event

    def on_window_resize(self, _event):
        self.canvas.draw_shapes()

if __name__ == "__main__":
    # Instantiate the Line Editor application and run the main loop
    app = LineEditorApp()
    app.mainloop()
```

Run the program and verify that the grid is drawn on the canvas. Also, verify that shapes are drawn in front of the grid. Try resizing the window to ensure that the grid is drawn to the canvas edges.



Snap to Grid

With the grid class in place, we will use its snap to grid methods to align shapes with the grid during draw, move, and resize operations.

The objectives of this section are:

- Modify the Mouse Class to snap coordinates to the grid for draw, move, and resize

- Note that the snap to grid method is in the Grid Class

Modify the Rectangle Class resize method to snap coordinates to the grid during resize operations. Call the `snap_to_grid()` method after any changes to the shape coordinates.

```
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Helper_Lib.point import Point

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 4 selectors
        self.s1, self.s2, self.s3, self.s4 = None, None, None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)

        if self.is_selected:
            self.draw_selectors()

    def create_selectors(self):
        # Calculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Create 4 selector objects: 4 corner of shape
        self.s1 = Selector(self.canvas, "s1", x1, y1)
        self.s2 = Selector(self.canvas, "s2", x2, y1)
        self.s3 = Selector(self.canvas, "s3", x2, y2)
        self.s4 = Selector(self.canvas, "s4", x1, y2)

        # Update the selector list
        self.sel_list = [self.s1, self.s2, self.s3, self.s4]

    def draw_selectors(self):
        # Recalculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points
```

```

# Define 5 selectors: 4 shape corner - number of selectors is unique to
each shape
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y1
    self.s3.x, self.s3.y = x2, y2
    self.s4.x, self.s4.y = x1, y2

# Draw the selectors
for s in self.sel_list:
    s.draw()

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        x2, y1 = self.canvas.grid.snap_to_grid(x2, y1)
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)
        self.x1, self.y2 = x1, y2

```

In a similar way, modify the Line Class resize method to call the `grid.snap_to_grid()` method whenever the line end coordinates are changed.

```
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector


class Line(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_line(self.points, fill="black", width=3)

        if self.is_selected:
            self.draw_selectors()

    def create_selectors(self):
        # Calculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Create 2 selector objects: 2 ends of the line
        self.s1 = Selector(self.canvas, "begin", x1, y1)
        self.s2 = Selector(self.canvas, "end", x2, y2)

        # Update the selector list
        self.sel_list = [self.s1, self.s2]

    def draw_selectors(self):
        # Recalculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Define 2 selectors: 2 ends of the line
        self.s1.x, self.s1.y = x1, y1
        self.s2.x, self.s2.y = x2, y2

        # Draw the selectors
        for s in self.sel_list:
            s.draw()

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
```

```

        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1

```

Modify the Mouse Class draw left down, draw left drag, move left down, move left drag, and resize left down methods to call `snap_to_grid` whenever coordinates are changed.

```

from Helper_Lib.point import Point
from Shape_Lib.rectangle import Rectangle
from Shape_Lib.line import Line


class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_obj = None
        self.current_shape = None
        self.current_shape_obj = None

        self.start = Point(0,0)
        self.offset1 = Point(0,0)
        self.offset2 = Point(0,0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def draw_bind_mouse_events(self):
        self.canvas.bind("<Button-1>", self.draw_left_down)
        self.canvas.bind("<B1-Motion>", self.draw_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

    def move_bind_mouse_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)

```

```

        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def draw_left_down(self, event):
        self.unselect_all_shapes()
        self.start.x = event.x
        self.start.y = event.y
        self.start.x, self.start.y =
self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "line":
            self.current_shape_obj = Line(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)

        if self.current_shape_obj is not None:
            self.canvas.shape_list.append(self.current_shape_obj)
            self.canvas.draw_shapes()

    def draw_left_drag(self, event):
        if self.current_shape_obj:
            x, y = event.x, event.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            self.current_shape_obj.x1, self.current_shape_obj.y1 =
self.start.x, self.start.y
            self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
            self.canvas.draw_shapes()

    def draw_left_up(self, _event):
        self.current_shape = None
        self.current_shape_obj = None
        self.unbind_mouse_events()
        self.move_bind_mouse_events()

    def move_left_down(self, event):
        if self.selected_obj:
            x, y = event.x, event.y
            sel = self.selected_obj.check_selector_hit(x, y)
            if sel:
                self.selected_obj.selector = sel.name
                self.unbind_mouse_events()
                self.bind_resize_mouse_events()
                self.resize_left_down(event)
                return
            else:

```

```

        a_shape = self.selected_obj
        a_shape.is_selected = False
        self.selected_obj = None
        self.canvas.draw_shapes()

    x, y = event.x, event.y
    self.select_shape(x, y)
    if self.selected_obj:
        x1, y1 = self.selected_obj.x1, self.selected_obj.y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        x2, y2 = self.selected_obj.x2, self.selected_obj.y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.offset1.x = x - x1
        self.offset1.y = y - y1
        self.offset2.x = x - x2
        self.offset2.y = y - y2
        self.canvas.draw_shapes()

def move_left_drag(self, event):
    if self.selected_obj:
        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.selected_obj.x1, self.selected_obj.y1 = x, y
        x = event.x - self.offset2.x
        y = event.y - self.offset2.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.selected_obj.x2, self.selected_obj.y2 = x, y
        self.canvas.draw_shapes()

def move_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0

def bind_resize_mouse_events(self):
    self.canvas.bind("<Button-1>", self.resize_left_down)
    self.canvas.bind("<B1-Motion>", self.resize_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

def resize_left_down(self, event):
    if self.selected_obj:
        x1, y1 = self.selected_obj.x1, self.selected_obj.y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        x2, y2 = self.selected_obj.x2, self.selected_obj.y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.offset1.x = event.x - x1

```

```

        self.offset1.y = event.y - y1
        self.offset2.x = event.x - x2
        self.offset2.y = event.y - y2
        self.canvas.draw_shapes()

    def resize_left_drag(self, event):
        if self.selected_obj:
            offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
self.offset2.y]
            self.selected_obj.resize(offsets, event)
            self.canvas.draw_shapes()

    def resize_left_up(self, _event):
        self.offset1.x, self.offset1.y = 0, 0
        self.offset2.x, self.offset2.y = 0, 0
        self.canvas.mouse.unbind_mouse_events()
        self.canvas.mouse.move_bind_mouse_events()

    def select_shape(self, x, y):
        self.unselect_all_shapes()
        for shape in self.canvas.shape_list:
            if shape.x1 <= x <= shape.x2 and shape.y1 <= y <= shape.y2:
                shape.is_selected = True
                self.selected_obj = shape
                self.canvas.draw_shapes()

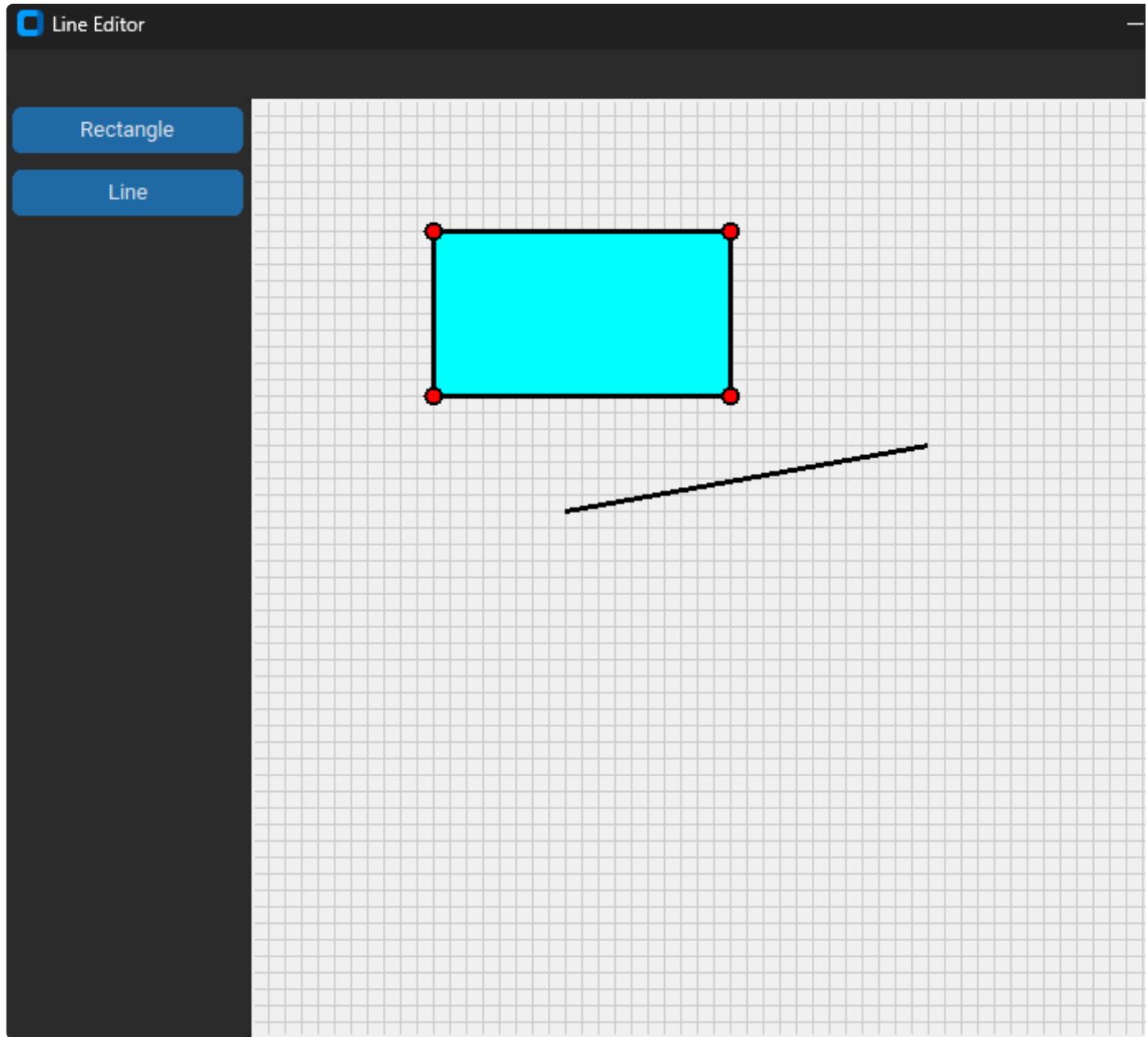
    def unselect_all_shapes(self):
        for shape in self.canvas.shape_list:
            shape.is_selected = False

        self.selected_obj = None
        self.canvas.draw_shapes()

```

Mouse Class lines of code: 149 lines

Run the program and verify that Shapes are drawn with snap to grid.



Shape to Line Connections

Next, we want to add the capability to attach a line to a shape such that if the shape moves or is resized, the attached line moves according. We need a way to identifying connection points called "connectors" on the shape when a line is drawn. Each shape will have a `line_list` to keep track of "connections" to lines. To formalize these new structural features, we will create two new classes that represent the connectors called `Connector` and the connections called `Connection`.

When a line is drawn, the Mouse Class will check to see if the beginning or end of the line is placed on a connector. If it has, a new connection will be added to the shape `line_list`.

Connectors will be drawn as "cyan" colored ovals. Each shape defines the position and number of connectors. Lines do not have connectors. Rectangles, ovals, triangle, text, and images can have connectors. Connector will generally be defined at the center of each side of the shape and at its center. This eliminate overlay between connectors and selectors.

This is the most difficult programming task for this project and many iterations were programmed before arriving at the solution. I think the key lesson learned is to "formalize" features into classes using the patterns of creation and drawing in the following code.

Connector Class

Similar to selectors, a connector represents a point on the side or center of a shape where a line can make a connection. The connector will be drawn as a "cyan" colored oval when the user is drawing or resizing a line.

The objectives of this section are:

- Create a new Connector Class
- Add a draw() method that draws a cyan oval
- Modify Rectangle Class to define connectors
- Modify Rectangle Class to draw connectors if Mouse Class `self.mode = "line_draw"`
- When in `"line_draw"` mode, show connectors on all other shapes
- In the Left Frame Class, set to line draw mode if Line is selected

Create a new class called Connector in a file called `connector.py`. The Connector Class attributes are canvas, name string, and its x, y position coordinates. The radius of a connector oval is set to 5 px. We add a `connector_hit_test` method that returns True if the event coordinates are within the connector boundaries. Add a draw() method that draws the connector with a fill color of "cyan", a border color of "black", and a border width of 2 px.

```
class Connector:  
    def __init__(self, canvas, name, x, y):  
        self.canvas = canvas  
        self.name = name  
        self.x = x  
        self.y = y  
  
        self.radius = 5
```

```

def conn_hit_test(self, event_x, event_y):
    x1, y1 = self.x - self.radius, self.y - self.radius
    x2, y2 = self.x + self.radius, self.y + self.radius
    if x1 <= event_x <= x2 and y1 <= event_y <= y2:
        return True
    else:
        return False

def draw(self):
    conn_points = [self.x - self.radius, self.y - self.radius, self.x +
    self.radius, self.y + self.radius]
    self.canvas.create_oval(conn_points, fill="cyan", outline="black",
width=2)

```

Modify the Rectangle Class to import the new Connector Class. After the code that creates 4 selectors, add code to create 5 connectors called c1, c2, c3, c4, c5. Add two new methods that create the connectors and draw the connectors. The create connector method creates the five connectors, sets the name and position and adds them to the connector list called `conn_list`. Note that we are following the same "pattern" that we used to create and draw selectors. The draw connectors list updates each connector position and draws the connector as a "cyan" oval at the connector position at the center of each side and one at the center of the Rectangle. If the mouse mode is set to `"line_draw"` string, the connectors are drawn.

```

from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector
from Helper_Lib.point import Point

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 4 selectors
        self.s1, self.s2, self.s3, self.s4 = None, None, None, None
        self.create_selectors()

        # Create 5 connectors
        self.c1, self.c2, self.c3, self.c4, self.c5 = None, None, None, None,

```

None

```
    self.create_connectors()

def draw(self):
    self.points = [self.x1, self.y1, self.x2, self.y2]
    self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)

    if self.is_selected:
        self.draw_selectors()

    if self.canvas.mouse.mode == "line_draw":
        self.draw_connectors()

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 4 selector objects: 4 corner of shape
    self.s1 = Selector(self.canvas, "s1", x1, y1)
    self.s2 = Selector(self.canvas, "s2", x2, y1)
    self.s3 = Selector(self.canvas, "s3", x2, y2)
    self.s4 = Selector(self.canvas, "s4", x1, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2, self.s3, self.s4]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 5 selectors: 4 shape corner - number of selectors is unique to
    each shape
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y1
    self.s3.x, self.s3.y = x2, y2
    self.s4.x, self.s4.y = x1, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def create_connectors(self):
    # Calculate position of connectors from current shape position and size
    x1, y1, x2, y2 = self.points
    w, h = x2 - x1, y2 - y1
```

```

        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 5 connectors: shape center, 4 side centers - number of
        # connectors is unique to each shape
        self.c1 = Connector(self.canvas, "c1", center.x, center.y)
        self.c2 = Connector(self.canvas, "c2", center.x, center.y - h / 2)
        self.c3 = Connector(self.canvas, "c3", center.x + w / 2, center.y)
        self.c4 = Connector(self.canvas, "c4", center.x, center.y + h / 2)
        self.c5 = Connector(self.canvas, "c5", center.x - w / 2, center.y)

        # Update the connector list
        self.conn_list = [self.c1, self.c2, self.c3, self.c4, self.c5]

    def draw_connectors(self):
        # Recalculate position of connectors from current shape position and
        size
        self.points = [self.x1, self.y1, self.x2, self.y2]
        x1, y1, x2, y2 = self.points
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 5 connectors: shape center, 4 side centers - number of
        # connectors is unique to each shape
        self.c1.x, self.c1.y = center.x, center.y
        self.c2.x, self.c2.y = center.x, center.y - h / 2
        self.c3.x, self.c3.y = center.x + w / 2, center.y
        self.c4.x, self.c4.y = center.x, center.y + h / 2
        self.c5.x, self.c5.y = center.x - w / 2, center.y

        # Draw the connectors
        for c in self.conn_list:
            c.draw()

    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - h/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
            self.x1, self.y1 = x1, y1

```

```

    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        x2, y1 = self.canvas.grid.snap_to_grid(x2, y1)
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)
        self.x1, self.y2 = x1, y2

```

Modify the Left Frame Class to set the mouse mode to `"line_draw"` when the Line Button is selected.

```

import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self,
                                    text="Rectangle",
                                    command=self.create_rectangle)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

        line_button = ctk.CTkButton(self,
                                    text="Line",
                                    command=self.create_line)
        line_button.pack(side=ctk.TOP, padx=5, pady=5)

    def create_rectangle(self):
        self.canvas.mouse.mode = None
        self.canvas.draw_shapes()
        self.canvas.mouse.current_shape = "rectangle"

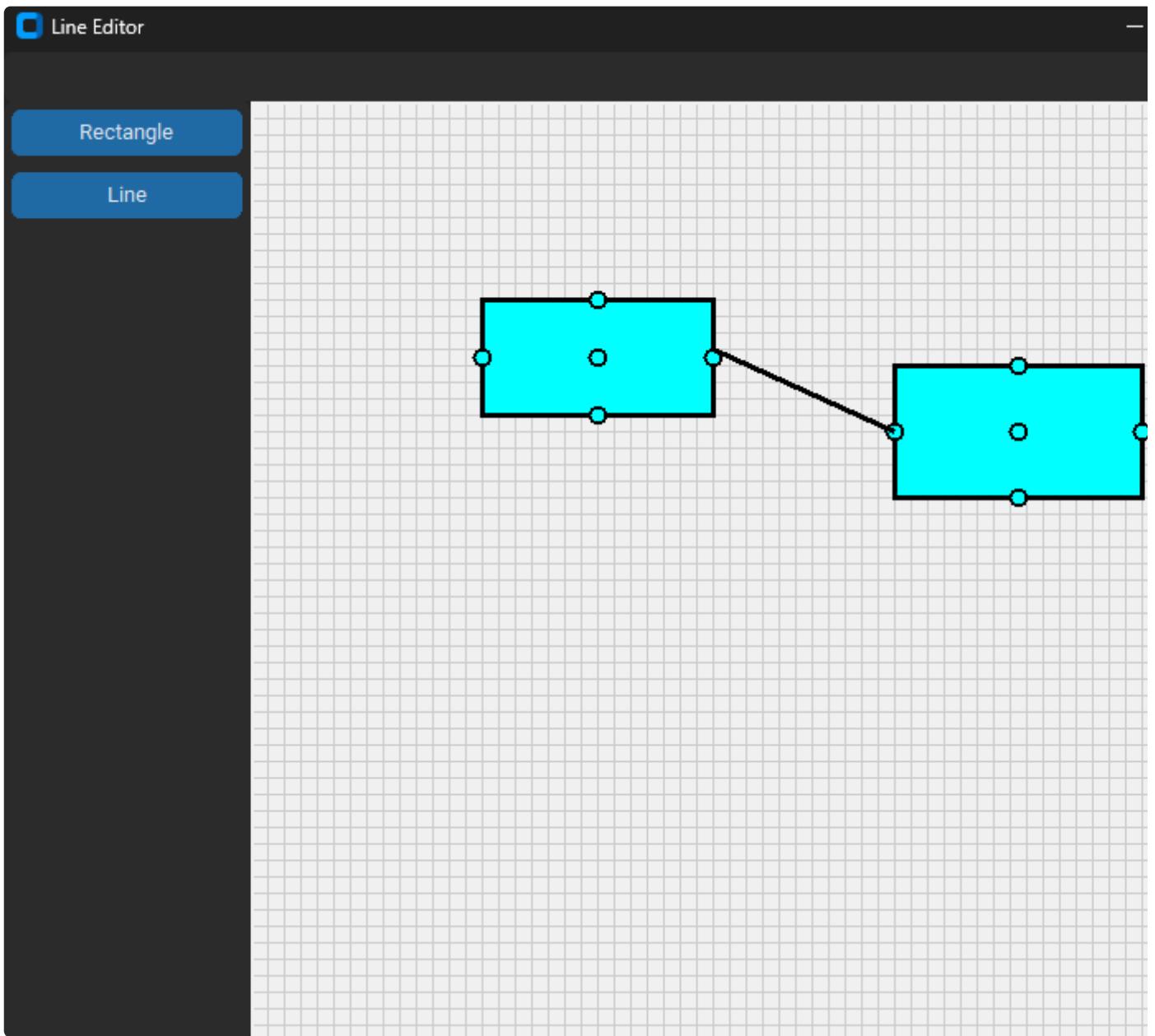
```

```
self.canvas.mouse.draw_bind_mouse_events()

def create_line(self):
    self.canvas.mouse.mode = "line_draw"
    self.canvas.draw_shapes()
    self.canvas.mouse.current_shape = "line"
    self.canvas.mouse.draw_bind_mouse_events()
```

Note that the Line Class has no connectors. Connectors are used on shapes to "connect" a line to it.

Run the program and verify that connectors are shown on rectangles after drawing two rectangles and one line.



We will deal with the Mouse Class in the next section.

Connections Class

We need a formal way to define a "connections" to a shape connector. The connections defines the line object, line end as "begin" or "end", and the line/connector position.

Connections are added to the shapes `line_list` so that we can move the line if the shape moves in the next section.

The objectives of this section are:

- Create a new class called Connection

- When a line is drawn, check to see if either end is over a shape connector
- If it is, add the line "connection" to the shape line list
- A connection has the line obj, line end ("begin", "end"), connection point (x ,y)
- Snap line end to the connector point

Create a new Connection Class in the Helper_Lib directory called connection.py. Initialize the connection with a line object, line end, and x, y position.

```
from Helper_Lib.point import Point

class Connection:
    def __init__(self, line_obj, line_end, x, y):
        self.line_obj = line_obj
        self.line_end = line_end      # "begin" or "end"
        self.line_pos = Point(x,y)
```

Modify the Mouse Class by importing the Connection Class. Add mode variable to the class attributes. In the draw left down method when a line is drawn add a call to a new select connector method with arguments current obj, "begin" end string, and x, y position. In the draw left up method, add a call to the new select connector method with arguments current obj, "end" end string, and x, y position.

Add a `select_connector()` method that iterates over all shapes in the canvas shape list. For each shape, call `check_connector_hit` method to see if one of the shapes connectors have been selected. If a connector is selected, check to see if the line end is "begin" or "end". Set the line end coordinates to the selected connector coordinates. Create a new Connection, add it to the selected shapes line list, and redraw all shapes to similar a line snap to connector center point.

mouse.py

```
from Helper_Lib.point import Point
from Helper_Lib.connection import Connection
from Shape_Lib.rectangle import Rectangle
from Shape_Lib.line import Line
```

```

class Mouse:

    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_obj = None
        self.current_shape = None
        self.current_shape_obj = None
        self.mode = None # Set to 'line_draw' to show shape connectors

        self.start = Point(0,0)
        self.offset1 = Point(0,0)
        self.offset2 = Point(0,0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def draw_bind_mouse_events(self):
        self.canvas.bind("<Button-1>", self.draw_left_down)
        self.canvas.bind("<B1-Motion>", self.draw_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

    def move_bind_mouse_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def draw_left_down(self, event):
        self.unselect_all_shapes()
        self.start.x = event.x
        self.start.y = event.y
        self.start.x, self.start.y =
        self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "line":
            self.current_shape_obj = Line(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

        if self.current_shape_obj is not None:
            self.canvas.shape_list.append(self.current_shape_obj)

```

```

        self.canvas.draw_shapes()

    def draw_left_drag(self, event):
        if self.current_shape_obj:
            x, y = event.x, event.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            self.current_shape_obj.x1, self.current_shape_obj.y1 =
self.start.x, self.start.y
            self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
            self.canvas.draw_shapes()

    def draw_left_up(self, event):
        if isinstance(self.current_shape_obj, Line):
            self.select_connector(self.current_shape_obj, "end", event.x,
event.y)
            self.current_shape = None
            self.current_shape_obj = None
            self.unbind_mouse_events()
            self.move_bind_mouse_events()

    def move_left_down(self, event):
        if self.selected_obj:
            x, y = event.x, event.y
            sel = self.selected_obj.check_selector_hit(x, y)
            if sel:
                self.selected_obj.selector = sel.name
                self.unbind_mouse_events()
                self.bind_resize_mouse_events()
                self.resize_left_down(event)
                return
            else:
                a_shape = self.selected_obj
                a_shape.is_selected = False
                self.selected_obj = None
                self.canvas.draw_shapes()

            x, y = event.x, event.y
            self.select_shape(x, y)
            if self.selected_obj:
                if not isinstance(self.selected_obj, Line):
                    self.mode = None
                x1, y1 = self.selected_obj.x1, self.selected_obj.y1
                x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
                x2, y2 = self.selected_obj.x2, self.selected_obj.y2
                x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
                self.offset1.x = x - x1

```

```

        self.offset1.y = y - y1
        self.offset2.x = x - x2
        self.offset2.y = y - y2
        self.canvas.draw_shapes()

    def move_left_drag(self, event):
        if self.selected_obj:
            x = event.x - self.offset1.x
            y = event.y - self.offset1.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            self.selected_obj.x1, self.selected_obj.y1 = x, y
            x = event.x - self.offset2.x
            y = event.y - self.offset2.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            self.selected_obj.x2, self.selected_obj.y2 = x, y
            self.canvas.draw_shapes()

    def move_left_up(self, _event):
        self.offset1.x, self.offset1.y = 0, 0
        self.offset2.x, self.offset2.y = 0, 0

    def bind_resize_mouse_events(self):
        self.canvas.bind("<Button-1>", self.resize_left_down)
        self.canvas.bind("<B1-Motion>", self.resize_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

    def resize_left_down(self, event):
        if self.selected_obj:
            if not isinstance(self.selected_obj, Line):
                self.mode = None
            x1, y1 = self.selected_obj.x1, self.selected_obj.y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
            x2, y2 = self.selected_obj.x2, self.selected_obj.y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
            self.offset1.x = event.x - x1
            self.offset1.y = event.y - y1
            self.offset2.x = event.x - x2
            self.offset2.y = event.y - y2
            self.canvas.draw_shapes()

    def resize_left_drag(self, event):
        if self.selected_obj:
            offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
self.offset2.y]
            self.selected_obj.resize(offsets, event)
            self.canvas.draw_shapes()

```

```

def resize_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0
    self.canvas.mouse.unbind_mouse_events()
    self.canvas.mouse.move_bind_mouse_events()

def select_shape(self, x, y):
    self.unselect_all_shapes()
    for shape in self.canvas.shape_list:
        if shape.x1 <= x <= shape.x2 and shape.y1 <= y <= shape.y2:
            shape.is_selected = True
            self.selected_obj = shape
            self.canvas.draw_shapes()

def unselect_all_shapes(self):
    for shape in self.canvas.shape_list:
        shape.is_selected = False

    self.selected_obj = None
    self.canvas.draw_shapes()

def select_connector(self, line_obj, line_end, x, y):
    for shape in self.canvas.shape_list:
        conn = shape.check_connector_hit(x, y)
        if conn:
            if line_end == "begin":
                line_obj.x1, line_obj.y1 = conn.x, conn.y
            elif line_end == "end":
                line_obj.x2, line_obj.y2 = conn.x, conn.y
            a_conn = Connection(line_obj, line_end, conn.x, conn.y)
            shape.line_list.append(a_conn)
            self.canvas.draw_shapes()
    return

```

Add a check connector hit method to the Shape Class that iterates over all connectors in the connector list and returns the connector is it is selected by calling each connectors `conn_hit_test` method.

```

class Shape:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1

```

```

        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.is_selected = False

        self.sel_list = []
        self.conn_list = []
        self.line_list = [] # Container for connected lines (connection)
        self.id = None

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def check_connector_hit(self, x, y): # Connector hit test method
        for conn in self.conn_list:
            if conn.conn_hit_test(x, y):
                return conn
        return None

```

In the Line Class, override the Shape `check_connector_hit` method with a call to pass. Lines do not have connectors.

```

from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector

class Line(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_line(self.points, fill="black", width=3)

```

```

    if self.is_selected:
        self.draw_selectors()

def check_connector_hit(self, x, y):
    pass

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 2 selector objects: 2 ends of the line
    self.s1 = Selector(self.canvas, "begin", x1, y1)
    self.s2 = Selector(self.canvas, "end", x2, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

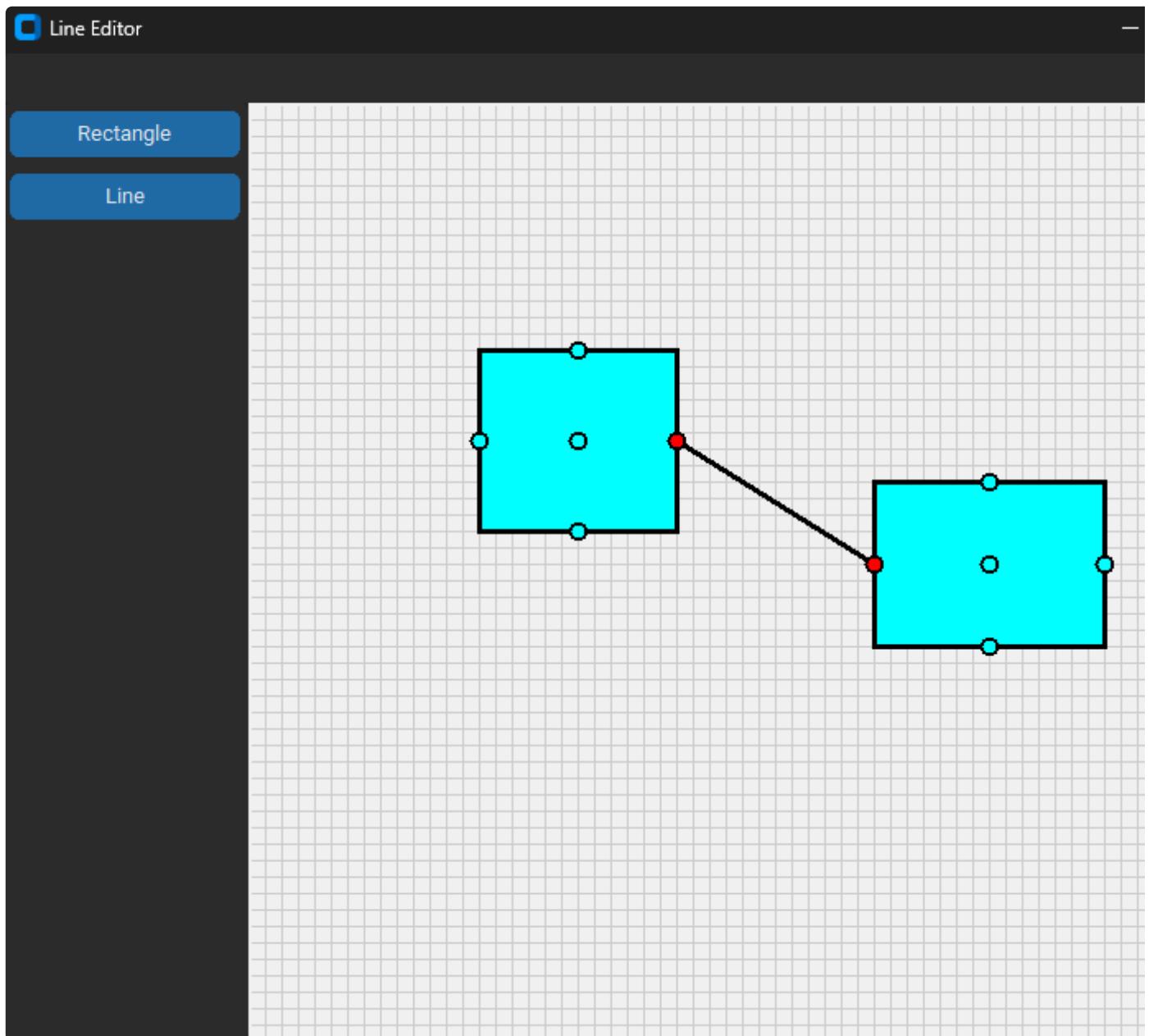
    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
        self.canvas.mouse.select_connector(self, "end", x2, y2)
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
        self.canvas.mouse.select_connector(self, "begin", x1, y1)

```

Run the program, draw two rectangles, and verify that you can draw a line between connectors.



Move Connected Lines if Shape Moves

When the user moves a shape with a connected line, we want the line to resize automatically so that visually it stays attached to the shape.

The objectives for this section are:

- Move connected lines if shape moves, resizes, or rotates
- Create a move method in each shape class that will move connected lines if attached

I put the `move_connection_lines()` method in the Shape Class. This method iterates over the shapes line list, it then iterates over each connector in the connector list. If a connector matches the connection connector, the connection line ends are set to the connector position which has moved.

For debug purposes, lets refactor the Connection Class to add a `__repr__` to print the connection parameters whenever a `print(connection)` is called.

```
class Connection:  
    def __init__(self, conn_obj, line_obj, line_end):  
        self.connector_obj = conn_obj  
        self.line_obj = line_obj  
        self.line_end = line_end      # "begin" or "end"  
  
    def __repr__(self):  
        return "Connection Object: " + self.connector_obj.name + \  
               " Connection Object Location: " + str(self.connector_obj.x) + ",  
" + str(self.connector_obj.y) + \  
               " Line Object Points: " + str(self.line_obj.points) + \  
               " Line End: " + self.line_end
```

Add a `__repr__()` method to the Connector Class for debug also.

```
class Connector:  
    def __init__(self, canvas, name, x, y):  
        self.canvas = canvas  
        self.name = name  
        self.x = x  
        self.y = y  
  
        self.radius = 5  
  
    def conn_hit_test(self, event_x, event_y):  
        x1, y1 = self.x - self.radius, self.y - self.radius  
        x2, y2 = self.x + self.radius, self.y + self.radius  
        if x1 <= event_x <= x2 and y1 <= event_y <= y2:  
            return True  
        else:  
            return False  
  
    def draw(self):  
        conn_points = [self.x - self.radius, self.y - self.radius, self.x +  
                      self.radius, self.y + self.radius]  
        self.canvas.create_oval(conn_points, fill="cyan", outline="black",  
                               width=2)  
  
    def __repr__(self):
```

```
        return "Connector " + self.name + " x: " + str(self.x) + " y: " +
str(self.y)
```

Modify the Rectangle Class to put the call to `move_connected_lines()` in the `draw_connectors()` method. Note that this eliminated calling it from move, resize, or rotate.

```
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector
from Helper_Lib.point import Point


class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        # Create 4 selectors
        self.s1, self.s2, self.s3, self.s4 = None, None, None, None
        self.create_selectors()

        # Create 5 connectors
        self.c1, self.c2, self.c3, self.c4, self.c5 = None, None, None, None,
None
        self.create_connectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill="cyan", outline="black",
width=3)

        if self.is_selected:
            self.draw_selectors()
            self.draw_connectors()

        if self.canvas.mouse.mode == "line_draw":
            self.draw_connectors()

    def create_selectors(self):
        # Calculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Create 4 selector objects: 4 corner of shape
```

```

        self.s1 = Selector(self.canvas, "s1", x1, y1)
        self.s2 = Selector(self.canvas, "s2", x2, y1)
        self.s3 = Selector(self.canvas, "s3", x2, y2)
        self.s4 = Selector(self.canvas, "s4", x1, y2)

        # Update the selector list
        self.sel_list = [self.s1, self.s2, self.s3, self.s4]

    def draw_selectors(self):
        # Recalculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Define 5 selectors: 4 shape corner - number of selectors is unique to
        # each shape
        self.s1.x, self.s1.y = x1, y1
        self.s2.x, self.s2.y = x2, y1
        self.s3.x, self.s3.y = x2, y2
        self.s4.x, self.s4.y = x1, y2

        # Draw the selectors
        for s in self.sel_list:
            s.draw()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.points
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 5 connectors: shape center, 4 side centers - number of
        # connectors is unique to each shape
        self.c1 = Connector(self.canvas, "c1", center.x, center.y)
        self.c2 = Connector(self.canvas, "c2", center.x, center.y - h / 2)
        self.c3 = Connector(self.canvas, "c3", center.x + w / 2, center.y)
        self.c4 = Connector(self.canvas, "c4", center.x, center.y + h / 2)
        self.c5 = Connector(self.canvas, "c5", center.x - w / 2, center.y)

        # Update the connector list
        self.conn_list = [self.c1, self.c2, self.c3, self.c4, self.c5]

    def draw_connectors(self):
        # Recalculate position of connectors from current shape position and
        # size
        self.points = [self.x1, self.y1, self.x2, self.y2]
        x1, y1, x2, y2 = self.points
        w, h = x2 - x1, y2 - y1

```

```

center = Point(x1 + w / 2, y1 + h / 2)

# Define 5 connectors: shape center, 4 side centers - number of
connectors is unique to each shape
self.c1.x, self.c1.y = center.x, center.y
self.c2.x, self.c2.y = center.x, center.y - h / 2
self.c3.x, self.c3.y = center.x + w / 2, center.y
self.c4.x, self.c4.y = center.x, center.y + h / 2
self.c5.x, self.c5.y = center.x - w / 2, center.y

# Draw the connectors
for c in self.conn_list:
    c.draw()

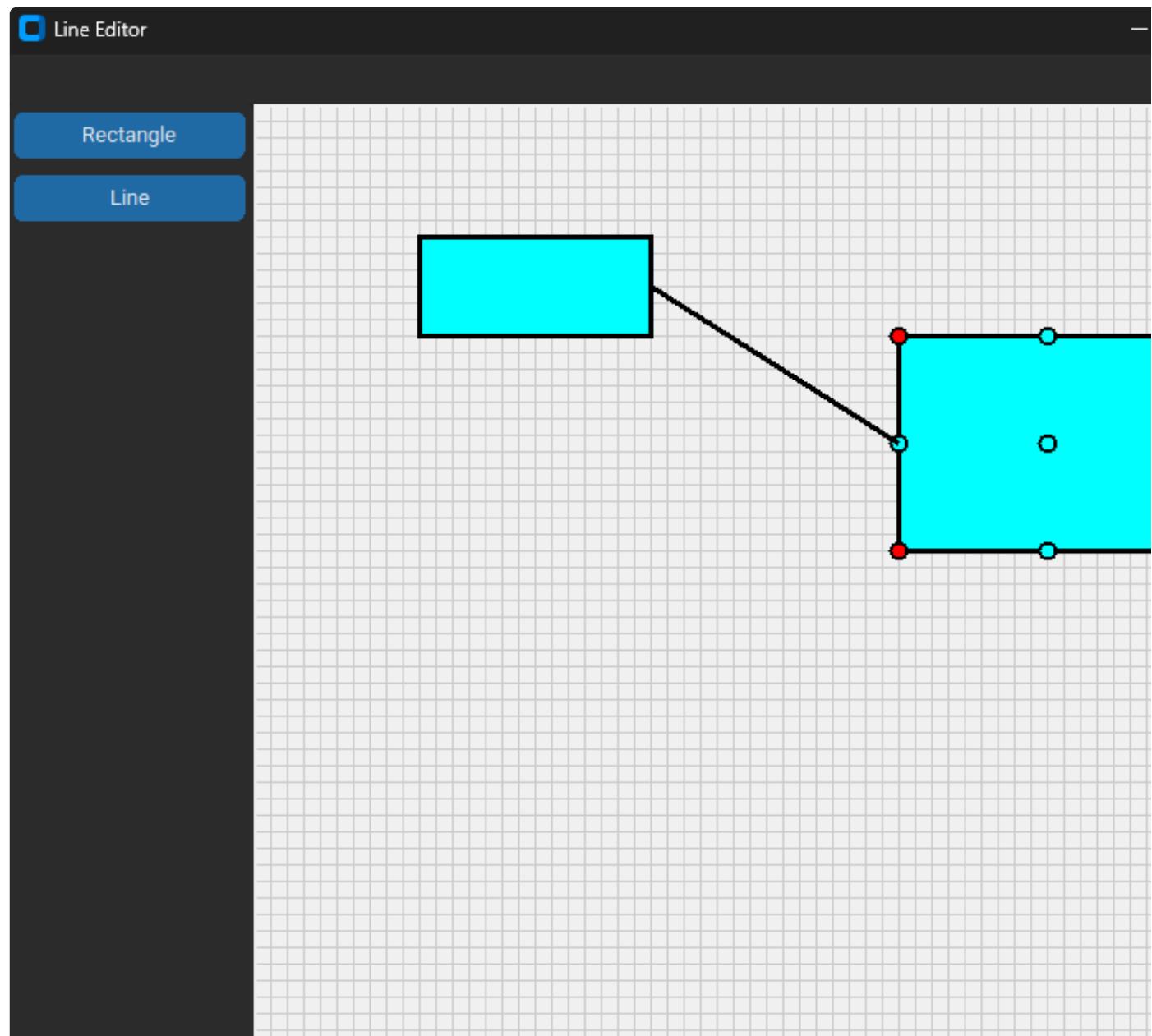
self.move_connected_lines()

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        x2, y1 = self.canvas.grid.snap_to_grid(x2, y1)
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)
        self.x1, self.y2 = x1, y2

```

Run the program, draw two rectangles and draw one line from a connector on the first rectangle to a connector on the second rectangle. Now, select and move one of the rectangles and confirm that the attached line resizes automatically to maintain the connection.



This took some refactoring to get it to work. The `__repr__` methods in the Connector and Connection classes were very useful to debug the code.

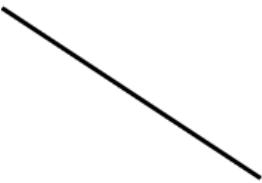
Other Line Types

In future projects, we will need other line types to create the type of diagram we need (schematics). We will add two line types: segmented line and elbow line. The following

diagram depicts the various line types. The segmented line has three(3) segments and the elbow line has two(2) segments. We will also need to control the starting direction of horizontal or vertical for the two new types.

Line Types

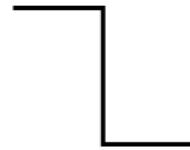
Straight Line



Elbow Line



Segmented Line



Segment Line

The first new line type is the Segment Line which has 3 segments.

The objectives for this section are:

- Draw a three segment line
- Create keyboard bindings to the "h" and "v" keys to set horizontal or vertical direction
- Create a button in the left frame called "Segment Line" to create the line
- Verify that the user can select, move, and resize the line
- Verify that the line ends move if a connected shape moves

Create a new Segment Line Class in a file called `segment_line.py` in the Shape_Lib directory. Import the Shape and Selector classes. In the class initializer, create a segment list variable which will hold the 3 segments of the line. Also create a line direction variable and set it to the canvas line direction variable. Create 2 selectors as we did with the straight line class. In the draw() method, set the points list to the line end coordinates, call the `create_segmented_line` method, and draw selectors if needed.

Add the create segmented line method which creates the 3 line segments based on the line direction variable. The new line segments are added to the shape segment list. Segments are drawn by the `draw_segments()` method which iterates over the segment list and draws the segments.

segment_line.py

```
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector


class SegmentLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None
        self.segment_list = None
        self.line_direction = self.canvas.line_direction

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.create_segmented_line()

        if self.is_selected:
            self.draw_selectors()

    def create_segmented_line(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        segment1, segment2, segment3 = None, None, None

        if self.line_direction == "horizontal":
            segment1 = self.x1, self.y1, self.x1 + w/2, self.y1
            segment2 = self.x1 + w/2, self.y1, self.x1 + w/2, self.y2
            segment3 = self.x1 + w/2, self.y2, self.x2, self.y2
        elif self.line_direction == "vertical":
            segment1 = self.x1, self.y1, self.x1, self.y1 + h/2
            segment2 = self.x1, self.y1 + h/2, self.x2, self.y1 + h/2
            segment3 = self.x2, self.y1 + h/2, self.x2, self.y2
        self.segment_list = [segment1, segment2, segment3]
        self.draw_segments()

    def draw_segments(self):
        for s in self.segment_list:
            self.canvas.create_line(s, fill="black", width=3)

    def check_connector_hit(self, x, y):
```

```

    pass

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 2 selector objects: 2 ends of the line
    self.s1 = Selector(self.canvas, "begin", x1, y1)
    self.s2 = Selector(self.canvas, "end", x2, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def move(self):
    pass

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
        self.canvas.mouse.select_connector(self, "end", x2, y2)
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
        self.canvas.mouse.select_connector(self, "begin", x1, y1)

def __repr__(self):
    return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,

```

```
y2: " + str(self.x2) + ", " + str(self.y2)
```

Add the `self.line_direction` attribute to the Canvas Class which determines the line initial direction. Add two new event handlers called `set_horizontal_line_direction` and `set_vertical_line_direction` which set the line direction variable if the 'h' or 'v' keys are pressed.

canvas.py

```
import customtkinter as ctk
from UI_Lib.mouse import Mouse
from Shape_Lib.line import Line
from Shape_Lib.grid import Grid

class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()
        self.shape_list = []
        self.mouse = Mouse(self)
        self.grid_size = 10
        self.grid = Grid(self, self.grid_size)
        self.grid.draw()
        self.line_direction = "horizontal"

    def draw_shapes(self):
        self.delete('all')
        self.grid.draw()
        for shape in self.shape_list:
            shape.draw()

    def rotate_shape(self, _event):
        if not isinstance(self.mouse.selected_obj, Line):
            self.mouse.selected_obj.rotate()
        self.draw_shapes()

    def set_horizontal_line_direction(self, _event):
        self.line_direction = "horizontal"

    def set_vertical_line_direction(self, _event):
        self.line_direction = "vertical"
```

Add the Segment Line menu option to Left Frame Class as we have done before.

```
import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self,
                                    text="Rectangle",
                                    command=self.create_rectangle)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

        line_button = ctk.CTkButton(self,
                                    text="Straight Line",
                                    command=self.create_line)
        line_button.pack(side=ctk.TOP, padx=5, pady=5)

        segment_line_button = ctk.CTkButton(self,
                                            text="Segment Line",
                                            command=self.create_segment_line)
        segment_line_button.pack(side=ctk.TOP, padx=5, pady=5)

    def create_rectangle(self):
        self.canvas.mouse.mode = None
        self.canvas.draw_shapes()
        self.canvas.mouse.current_shape = "rectangle"
        self.canvas.mouse.draw_bind_mouse_events()

    def create_line(self):
        self.canvas.mouse.mode = "line_draw"
        self.canvas.draw_shapes()
        self.canvas.mouse.current_shape = "line"
        self.canvas.mouse.draw_bind_mouse_events()

    def create_segment_line(self):
        self.canvas.mouse.mode = "line_draw"
        self.canvas.draw_shapes()
        self.canvas.mouse.current_shape = "segment"
        self.canvas.mouse.draw_bind_mouse_events()
```

In the Mouse Class, add the code to draw a segment line to Draw left down method.

mouse.py

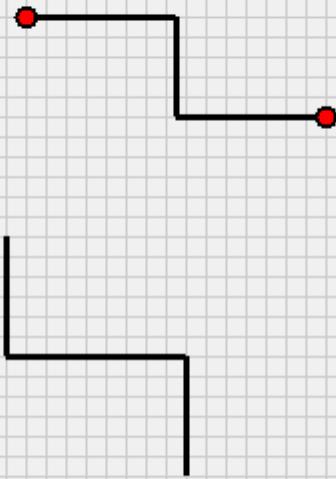
```
def draw_left_down(self, event):
    self.unselect_all_shapes()
    self.start.x = event.x
    self.start.y = event.y
    self.start.x, self.start.y =
    self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
    self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "line":
        self.current_shape_obj = Line(self.canvas, self.start.x,
    self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
    self.start.x, self.start.y)
    elif self.current_shape == "segment":
        self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
    self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
    self.start.x, self.start.y)

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)
        self.canvas.draw_shapes()
```

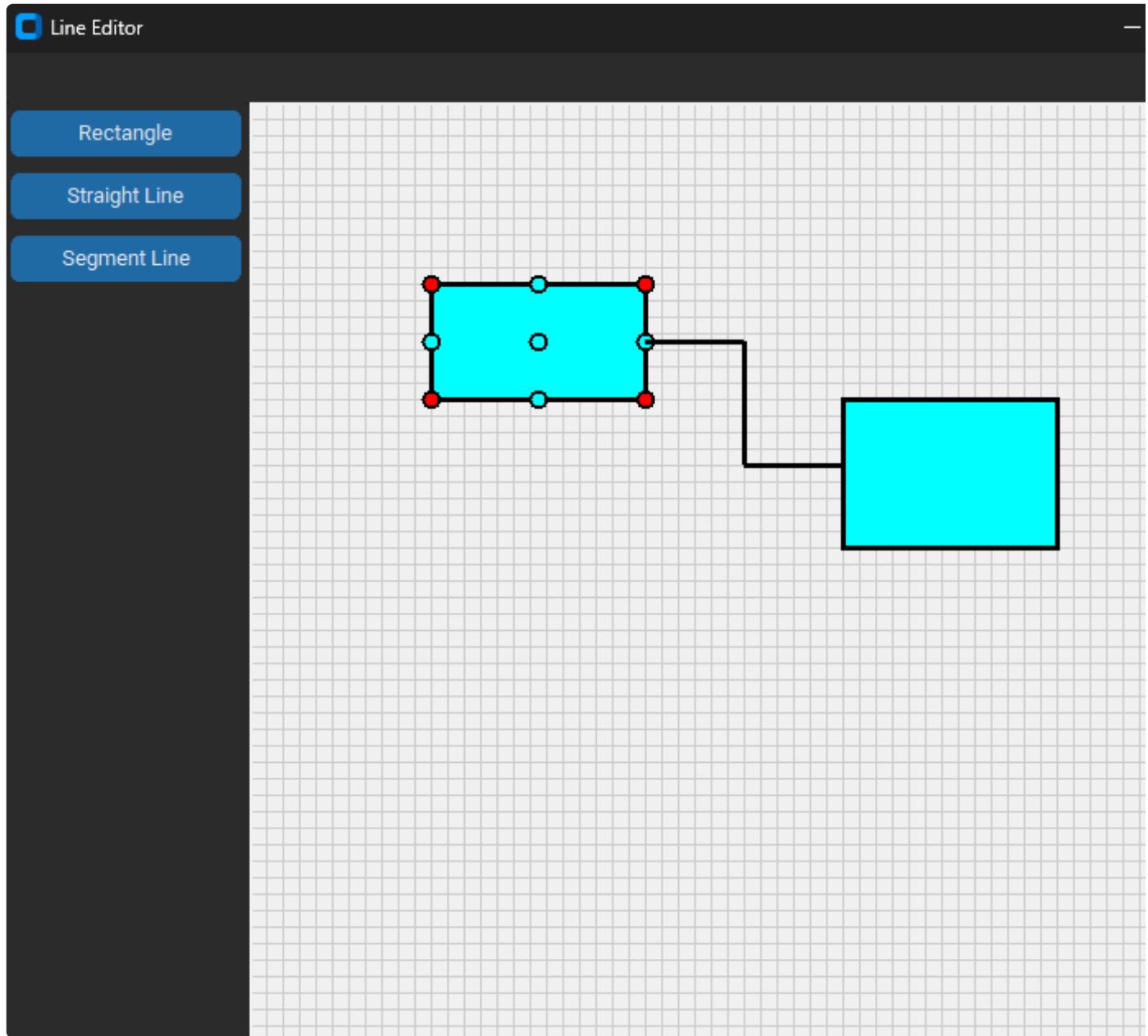
Mouse class lines of code: 177 lines

Run the program and confirm that you can create a segment line from the left menu. Try changing the initial direction using the "h" and "v" keys

Rectangle Straight Line Segment Line

To confirm that segment lines move when attached to a shape, create a diagram with two rectangles. Draw a Segment Line from a connector on the first rectangle to a connector on a second rectangle. Move one of the rectangles and confirm the the attached segment line

automatically resizes.



Elbow Line Class

This line type called Elbow Line is a two segment line.

The objectives of this section are:

- Draw a elbow (L-shaped) line
- Create a button in the left frame called "Elbow Line" to create the line
- Verify that the user can select, move, and resize the line
- Verify that the line ends move if a connected shape moves

Create a Elbow Line Class as we did for the Segment Line. The main difference is the `create_elbow_line` method which define 2 line segments based on the `line_direction` variable.

```
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector

class ElbowLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None
        self.segment_list = None
        self.line_direction = self.canvas.line_direction

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.create_elbow_line()

        if self.is_selected:
            self.draw_selectors()

    def create_elbow_line(self):
        segment1, segment2 = None, None
        if self.line_direction == "horizontal":
            segment1 = self.x1, self.y1, self.x2, self.y1
            segment2 = self.x2, self.y1, self.x2, self.y2
        elif self.line_direction == "vertical":
            segment1 = self.x1, self.y1, self.x1, self.y2
            segment2 = self.x1, self.y2, self.x2, self.y2
        self.segment_list = [segment1, segment2]
        self.draw_segments()

    def draw_segments(self):
        for s in self.segment_list:
            self.canvas.create_line(s, fill="black", width=3)

    def check_connector_hit(self, x, y):
        pass

    def create_selectors(self):
```

```

# Calculate position of selectors from current shape position
x1, y1, x2, y2 = self.points

# Create 2 selector objects: 2 ends of the line
self.s1 = Selector(self.canvas, "begin", x1, y1)
self.s2 = Selector(self.canvas, "end", x2, y2)

# Update the selector list
self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def move(self):
    pass

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
        self.canvas.mouse.select_connector(self, "end", x2, y2)
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
        self.canvas.mouse.select_connector(self, "begin", x1, y1)

def __repr__(self):
    return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

Add a new menu button called Elbow Line in the Left Frame Class that creates an elbow line.

```
import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self,
                                    text="Rectangle",
                                    command=self.create_rectangle)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

        line_button = ctk.CTkButton(self,
                                    text="Straight Line",
                                    command=self.create_line)
        line_button.pack(side=ctk.TOP, padx=5, pady=5)

        segment_line_button = ctk.CTkButton(self,
                                            text="Segment Line",
                                            command=self.create_segment_line)
        segment_line_button.pack(side=ctk.TOP, padx=5, pady=5)

        elbow_line_button = ctk.CTkButton(self,
                                         text="Elbow Line",
                                         command=self.create_elbow_line)
        elbow_line_button.pack(side=ctk.TOP, padx=5, pady=5)

    def create_rectangle(self):
        self.canvas.mouse.mode = None
        self.canvas.draw_shapes()
        self.canvas.mouse.current_shape = "rectangle"
        self.canvas.mouse.draw_bind_mouse_events()

    def create_line(self):
        self.canvas.mouse.mode = "line_draw"
        self.canvas.draw_shapes()
        self.canvas.mouse.current_shape = "line"
        self.canvas.mouse.draw_bind_mouse_events()
```

```

def create_segment_line(self):
    self.canvas.mouse.mode = "line_draw"
    self.canvas.draw_shapes()
    self.canvas.mouse.current_shape = "segment"
    self.canvas.mouse.draw_bind_mouse_events()

def create_elbow_line(self):
    self.canvas.mouse.mode = "line_draw"
    self.canvas.draw_shapes()
    self.canvas.mouse.current_shape = "elbow"
    self.canvas.mouse.draw_bind_mouse_events()

```

In the Mouse Class, add the code to draw an elbow line to Draw left down method.

mouse.py

```

def draw_left_down(self, event):
    self.unselect_all_shapes()
    self.start.x = event.x
    self.start.y = event.y
    self.start.x, self.start.y =
    self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "line":
        self.current_shape_obj = Line(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
    elif self.current_shape == "segment":
        self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
    elif self.current_shape == "elbow":
        self.current_shape_obj = ElbowLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

    if self.current_shape_obj is not None:

```

```

        self.canvas.shape_list.append(self.current_shape_obj)
        self.canvas.draw_shapes()

    def draw_left_drag(self, event):
        if self.current_shape_obj:
            x, y = event.x, event.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            self.current_shape_obj.x1, self.current_shape_obj.y1 =
self.start.x, self.start.y
            self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
            self.canvas.draw_shapes()

    def draw_left_up(self, event):
        if isinstance(self.current_shape_obj, Line) or
isinstance(self.current_shape_obj, SegmentLine) or \
            isinstance(self.current_shape_obj, ElbowLine):
            self.select_connector(self.current_shape_obj, "end", event.x,
event.y)
            self.current_shape = None
            self.current_shape_obj = None
            self.unbind_mouse_events()
            self.move_bind_mouse_events()

```

Mouse method lines of code: 182 lines

Run the program and draw elbow lines from the Left Menu. Try setting the starting direction using the 'h' and 'v' keys.



Line Editor

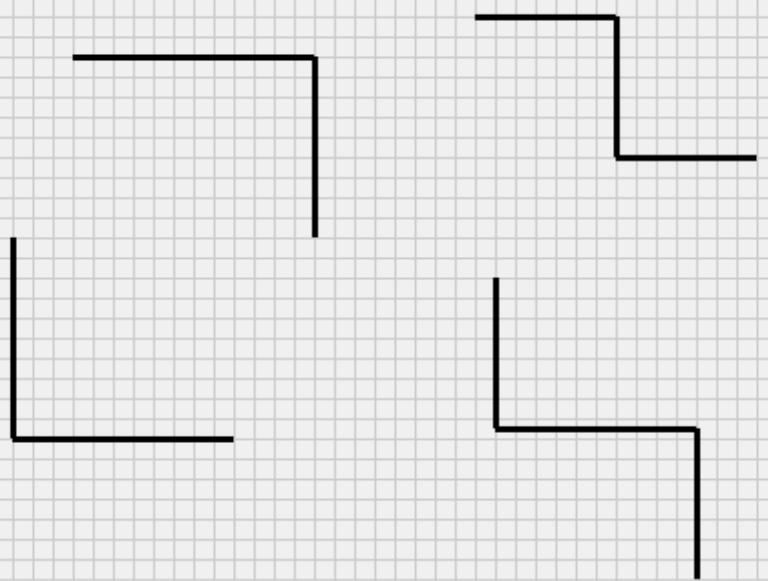
-

Rectangle

Straight Line

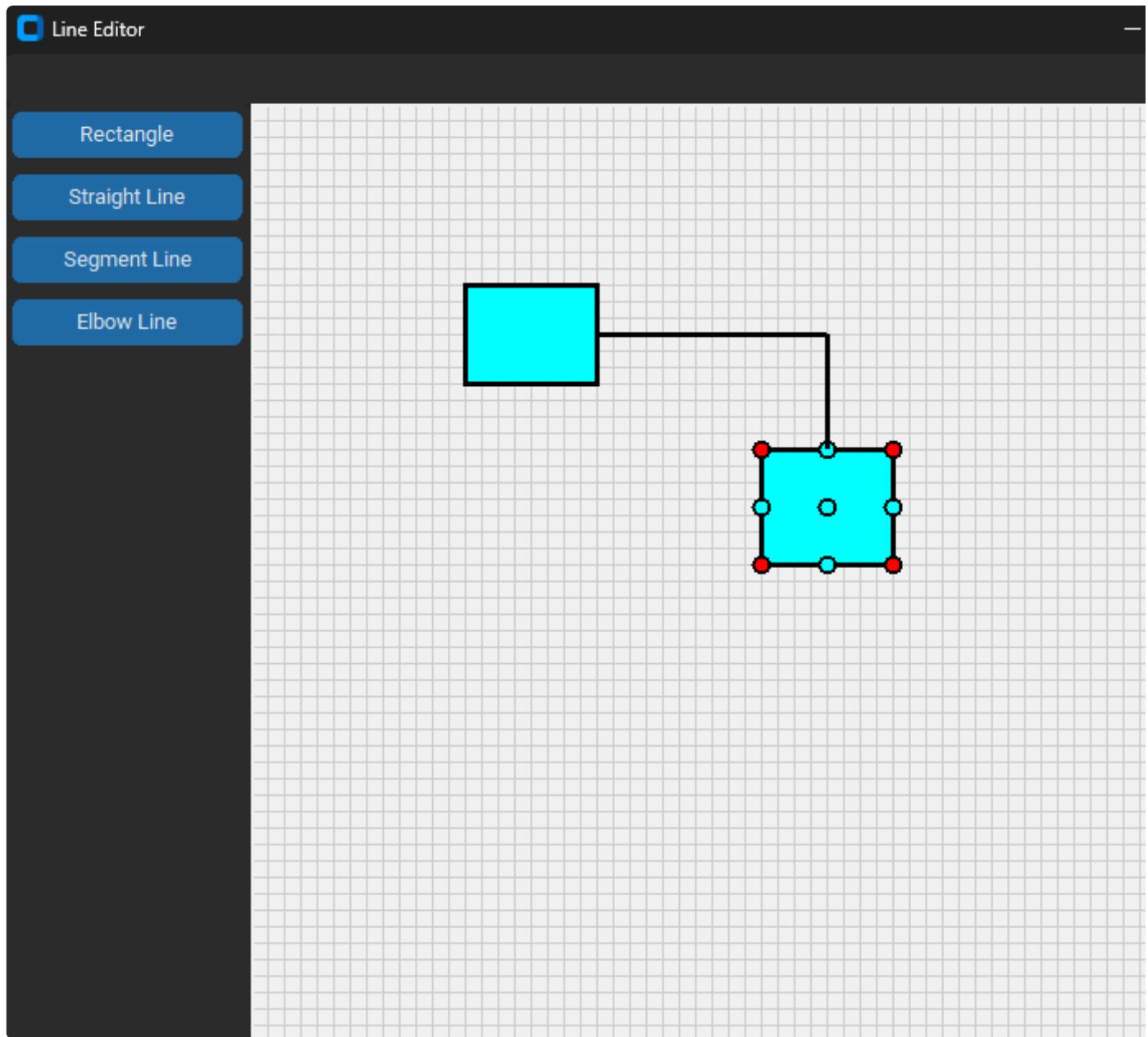
Segment Line

Elbow Line



Create a diagram with two rectangles and one elbow line drawn from a connector on the first rectangle to a connector on the second rectangle. Select and move one of the

rectangles to confirm that the elbow line automatically resizes.



Top Frame Class

The Top Frame Class adds shape appearance controls to control the appearance of a selected shape such as fill color, border color, and border width. It will also have grid controls that control the visibility of the grid and the grid size.

The objectives of this section are:

- Create a Top Frame class
- Add controls to change shape fill and border color as well as border width
- Add grid on/off and grid size controls

Create a custom Top Frame Class with all required controls.

```
import customtkinter as ctk
from CTkColorPicker import *

class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add top frame widgets here
        self.fill_color_button = ctk.CTkButton(self,
                                                text="Fill Color",
                                                command=self.set_fill_color)
        self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

        self.border_color_button = ctk.CTkButton(self,
                                                text="Border Color",
                                                command=self.set_border_color)
        self.border_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

        border_width_optionmenu = ctk.CTkOptionMenu(self,
                                                    values=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
                                                    command=self.set_border_width)
        border_width_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
        border_width_optionmenu.set("3")

        self.switch_var = ctk.StringVar(value="on")
        switch = ctk.CTkSwitch(self, text="Grid Visible",
                               command=self.grid_visibility,
                               variable=self.switch_var,
                               onvalue="on", offvalue="off")
        switch.pack(side=ctk.LEFT, padx=3, pady=3)

        grid_size_optionmenu = ctk.CTkOptionMenu(self,
                                                values=[5, 10, 15, 20, 25, 40, 50,
                                                       60, 70, 80, 90, 100],
                                                command=self.set_grid_size)
        grid_size_optionmenu.pack(side=ctk.LEFT, padx=3, pady=3)
        grid_size_optionmenu.set("10")

    def set_fill_color(self):
        if self.canvas.mouse.selected_obj:
```

```

        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.canvas.mouse.selected_obj.fill_color = color
        self.canvas.draw_shapes()

    def set_border_color(self):
        if self.canvas.mouse.selected_obj:
            pick_color = AskColor() # open the color picker
            color = pick_color.get() # get the color string
            self.canvas.mouse.selected_obj.border_color = color
            self.canvas.draw_shapes()

    def set_border_width(self, choice):
        if self.canvas.mouse.selected_obj:
            self.canvas.mouse.selected_obj.border_width = choice
            self.canvas.draw_shapes()

    def grid_visibility(self):
        if self.switch_var.get() == "on":
            self.canvas.grid.grid_visible = True
        elif self.switch_var.get() == "off":
            self.canvas.grid.grid_visible = False
        self.canvas.draw_shapes()

    def set_grid_size(self, choice):
        self.canvas.grid.grid_size = int(choice)
        self.canvas.draw_shapes()

```

Modify the Rectangle Class such that the fill color, border color, and border width can be set externally. This requires initialization of new variables and modification of the draw() method to use the new variables.

```

from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector
from Helper_Lib.point import Point

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

```

```

        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        # Create 4 selectors
        self.s1, self.s2, self.s3, self.s4 = None, None, None, None
        self.create_selectors()

        # Create 5 connectors
        self.c1, self.c2, self.c3, self.c4, self.c5 = None, None, None, None,
None
        self.create_connectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.create_rectangle(self.points, fill=self.fill_color,
outline=self.border_color,
                                         width=self.border_width)

        if self.is_selected:
            self.draw_selectors()
            self.draw_connectors()

        if self.canvas.mouse.mode == "line_draw":
            self.draw_connectors()

    def create_selectors(self):
        # Calculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Create 4 selector objects: 4 corner of shape
        self.s1 = Selector(self.canvas, "s1", x1, y1)
        self.s2 = Selector(self.canvas, "s2", x2, y1)
        self.s3 = Selector(self.canvas, "s3", x2, y2)
        self.s4 = Selector(self.canvas, "s4", x1, y2)

        # Update the selector list
        self.sel_list = [self.s1, self.s2, self.s3, self.s4]

    def draw_selectors(self):
        # Recalculate position of selectors from current shape position
        x1, y1, x2, y2 = self.points

        # Define 5 selectors: 4 shape corner - number of selectors is unique to
each shape
        self.s1.x, self.s1.y = x1, y1

```

```

        self.s2.x, self.s2.y = x2, y1
        self.s3.x, self.s3.y = x2, y2
        self.s4.x, self.s4.y = x1, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def create_connectors(self):
    # Calculate position of connectors from current shape position and size
    x1, y1, x2, y2 = self.points
    w, h = x2 - x1, y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    # Define 5 connectors: shape center, 4 side centers - number of
    # connectors is unique to each shape
    self.c1 = Connector(self.canvas, "c1", center.x, center.y)
    self.c2 = Connector(self.canvas, "c2", center.x, center.y - h / 2)
    self.c3 = Connector(self.canvas, "c3", center.x + w / 2, center.y)
    self.c4 = Connector(self.canvas, "c4", center.x, center.y + h / 2)
    self.c5 = Connector(self.canvas, "c5", center.x - w / 2, center.y)

    # Update the connector list
    self.conn_list = [self.c1, self.c2, self.c3, self.c4, self.c5]

def draw_connectors(self):
    # Recalculate position of connectors from current shape position and
    # size
    self.points = [self.x1, self.y1, self.x2, self.y2]
    x1, y1, x2, y2 = self.points
    w, h = x2 - x1, y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    # Define 5 connectors: shape center, 4 side centers - number of
    # connectors is unique to each shape
    self.c1.x, self.c1.y = center.x, center.y
    self.c2.x, self.c2.y = center.x, center.y - h / 2
    self.c3.x, self.c3.y = center.x + w / 2, center.y
    self.c4.x, self.c4.y = center.x, center.y + h / 2
    self.c5.x, self.c5.y = center.x - w / 2, center.y

    # Draw the connectors
    for c in self.conn_list:
        c.draw()

    self.move_connected_lines()

```

```

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        x2, y1 = self.canvas.grid.snap_to_grid(x2, y1)
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)
        self.x1, self.y2 = x1, y2

```

Similarly, modify the Line Class to set the fill color and border width. Lines do no have a border color.

```

from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector

class Line(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None

        self.fill_color = "black"

```

```

    self.border_width = 3

    # Create 2 selectors
    self.s1, self.s2 = None, None
    self.create_selectors()

def draw(self):
    self.points = [self.x1, self.y1, self.x2, self.y2]
    self.canvas.create_line(self.points, fill=self.fill_color,
width=self.border_width)

    if self.is_selected:
        self.draw_selectors()

def check_connector_hit(self, x, y):
    pass

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 2 selector objects: 2 ends of the line
    self.s1 = Selector(self.canvas, "begin", x1, y1)
    self.s2 = Selector(self.canvas, "end", x2, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

def move(self):
    pass

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":

```

```

        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
        self.canvas.mouse.select_connector(self, "end", x2, y2)
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
        self.canvas.mouse.select_connector(self, "begin", x1, y1)

    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

Modify the Segment Line Class to set the fill color and border width.

```

from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector

class SegmentLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None
        self.segment_list = None
        self.line_direction = self.canvas.line_direction

        self.fill_color = "black"
        self.border_width = 3

        # Create 2 selectors
        self.s1, self.s2 = None, None
        self.create_selectors()

    def draw(self):
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.create_segmented_line()

        if self.is_selected:
            self.draw_selectors()

```

```

def create_segmented_line(self):
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    segment1, segment2, segment3 = None, None, None

    if self.line_direction == "horizontal":
        segment1 = self.x1, self.y1, self.x1 + w/2, self.y1
        segment2 = self.x1 + w/2, self.y1, self.x1 + w/2, self.y2
        segment3 = self.x1 + w/2, self.y2, self.x2, self.y2
    elif self.line_direction == "vertical":
        segment1 = self.x1, self.y1, self.x1, self.y1 + h/2
        segment2 = self.x1, self.y1 + h/2, self.x2, self.y1 + h/2
        segment3 = self.x2, self.y1 + h/2, self.x2, self.y2
    self.segment_list = [segment1, segment2, segment3]
    self.draw_segments()

def draw_segments(self):
    for s in self.segment_list:
        self.canvas.create_line(s, fill=self.fill_color,
width=self.border_width)

def check_connector_hit(self, x, y):
    pass

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 2 selector objects: 2 ends of the line
    self.s1 = Selector(self.canvas, "begin", x1, y1)
    self.s2 = Selector(self.canvas, "end", x2, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors
    for s in self.sel_list:
        s.draw()

```

```

def move(self):
    pass

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.x2, self.y2 = x2, y2
        self.canvas.mouse.select_connector(self, "end", x2, y2)
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        self.x1, self.y1 = x1, y1
        self.canvas.mouse.select_connector(self, "begin", x1, y1)

def __repr__(self):
    return "Line: x1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

Modify the Elbow Line Class to set the fill color and border width.

```

from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector

class ElbowLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.selector = None
        self.segment_list = None
        self.line_direction = self.canvas.line_direction

        self.fill_color = "black"
        self.border_width = 3

    # Create 2 selectors
    self.s1, self.s2 = None, None
    self.create_selectors()

```

```

def draw(self):
    self.points = [self.x1, self.y1, self.x2, self.y2]
    self.create_elbow_line()

    if self.is_selected:
        self.draw_selectors()

def create_elbow_line(self):
    segment1, segment2 = None, None
    if self.line_direction == "horizontal":
        segment1 = self.x1, self.y1, self.x2, self.y1
        segment2 = self.x2, self.y1, self.x2, self.y2
    elif self.line_direction == "vertical":
        segment1 = self.x1, self.y1, self.x1, self.y2
        segment2 = self.x1, self.y2, self.x2, self.y2
    self.segment_list = [segment1, segment2]
    self.draw_segments()

def draw_segments(self):
    for s in self.segment_list:
        self.canvas.create_line(s, fill=self.fill_color,
width=self.border_width)

def check_connector_hit(self, x, y):
    pass

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Create 2 selector objects: 2 ends of the line
    self.s1 = Selector(self.canvas, "begin", x1, y1)
    self.s2 = Selector(self.canvas, "end", x2, y2)

    # Update the selector list
    self.sel_list = [self.s1, self.s2]

def draw_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.points

    # Define 2 selectors: 2 ends of the line
    self.s1.x, self.s1.y = x1, y1
    self.s2.x, self.s2.y = x2, y2

    # Draw the selectors

```

```

        for s in self.sel_list:
            s.draw()

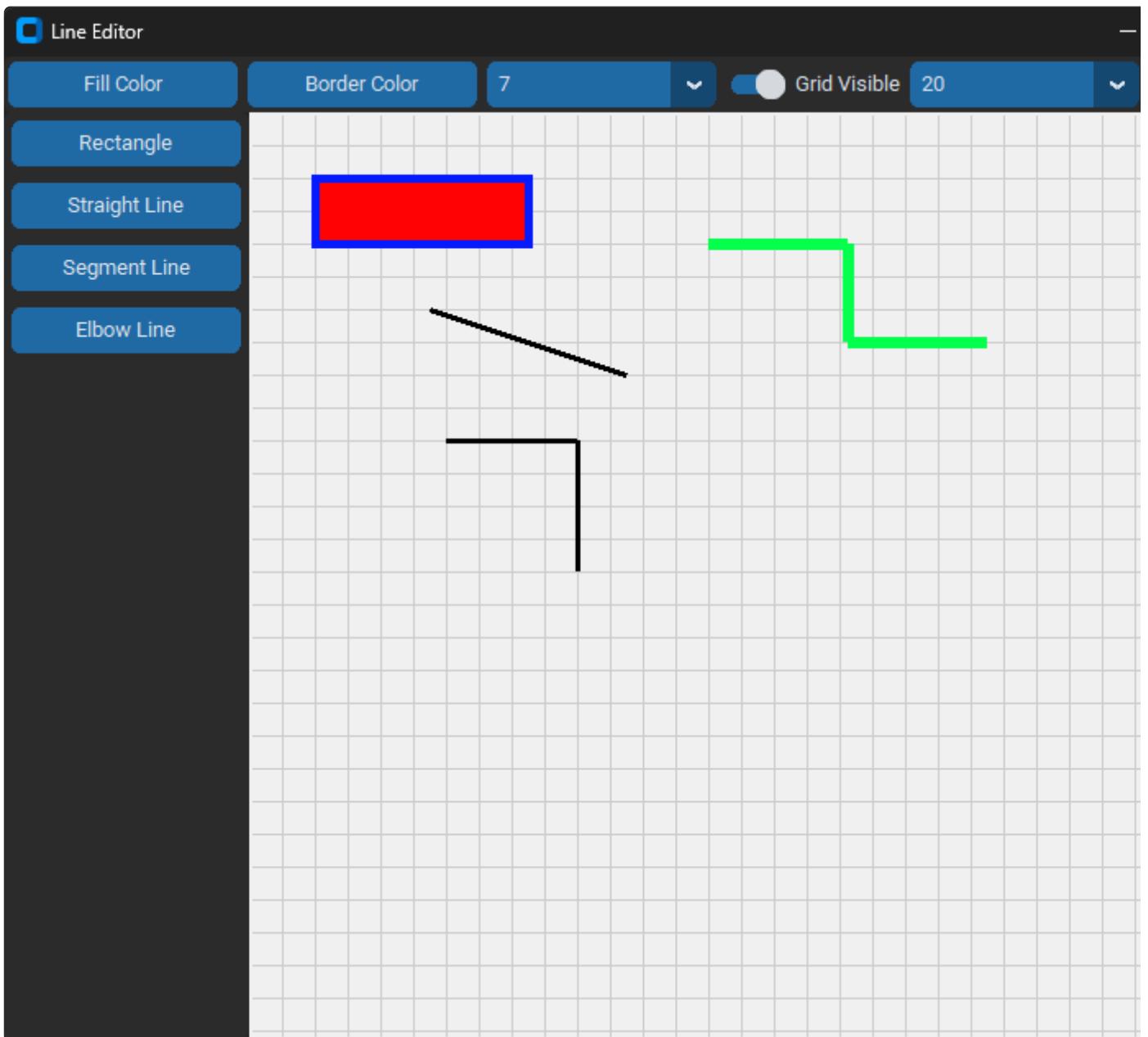
    def move(self):
        pass

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
            self.x2, self.y2 = x2, y2
            self.canvas.mouse.select_connector(self, "end", x2, y2)
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
            self.x1, self.y1 = x1, y1
            self.canvas.mouse.select_connector(self, "begin", x1, y1)

    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

Run the program and draw the four shapes and confirm that the user can change the appearance of a selected shape such as fill color, border color, and border width. Also, test the grid visible switch and grid size option menu.



Summary

This concludes the Line Editor project. Congratulate yourself if you completed the project. If you still have problems, the complete source code is available on GitHub at [GitHub](#).

With the Shape Editor and Line Editor projects under our belt, we are ready to move to the advanced project section. As you will see, we will reuse much of the Shape Editor and Line Editor to create the first advanced project called Diagram Editor where we will create an application with commercial application features such as files, help and configuration menu options.

Chapter 5 - Diagram Editor

The first advanced project is a Diagram Editor which will integrate the techniques and solutions that were developed in the Shape Editor and Line Editor applications.

The Diagram Editor is a diagram creation application that allows the user to select shapes from a menu which creates the shape on the canvas. The user can draw, move, rotate, and resize the shape as well as change the fill color, border color, or border width.

Program Design & Architecture:

- ✓ Object-Oriented Programming (OOP) - class abstraction, encapsulation, inheritance, & polymorphism
- ✓ Modules - separate files from the main program file
- ✓ Package - directory with `__init__.py` file that imports all directory modules such that the program can use a standard import statement

Libraries

- ✓ Helper Library (Helper_Lib) - Utility classes or other modules that add helpful features
- ✓ Shape Library (ShapeLib) - *All shape classes that wrap the `canvas.create()` function where `* is Rectangle, Oval, Polygon, Line, Text, Image, etc.
- ✓ User Interface Library (UI_Lib) - GUI interface classes for custom frames, canvas, keyboard, mouse, etc.

Shape Library

- ✓ Shape Class - base class for shape classes
- ✓ Rectangle Class
- ✓ Oval Class
- ✓ Triangle Class
- ✓ Straight Line Class
- ✓ Segment Line Class
- ✓ Elbow Line Class
- ✓ Text Class

- ✓ Picture Class
- ✓ Connector Class
- ✓ Selector Class
- ✓ Connection Class
- ✓ Grid Class
 - ✓ Snap to Grid
 - ✓ Grid visibility on/off
 - ✓ Snap size
 - ✓ Grid size

Shape Modification

- ✓ Keyboard rotate
- ✓ Set fill color
- ✓ Set border color
- ✓ Set border width
- ✓ Mouse draw
- ✓ Mouse move
- ✓ Mouse resize

User Interface Library

- ✓ Canvas Class
- ✓ Mouse Class
 - ✓ Mouse draw
 - ✓ Mouse move
 - ✓ Mouse resize
- ✓ Keyboard Class
 - ✓ Keyboard rotate
- ✓ Top Frame Class
 - ✓ File Menu Frame
 - ✓ Settings Frame
 - ✓ Help Frame
 - ✓ Shape Appearance Frame
 - ✓ Set fill color
 - ✓ Set border color
 - ✓ Set border width

- ✓ Left Frame Class
 - ✓ Shape Button Frame
 - ✓ Line Button Frame
- ✓ Bottom Frame Class
 - ✓ Message label

Helper Library

- ✓ Point Class

Image Library - not a python package

- ✓ Create images in Microsoft PowerPoint
- ✓ Adjust image dimension in Microsoft Paint 3D
- ✓ Use the `expand=True` setting in `image.rotate()` method call to avoid drawing a new image for each rotation angle
- ✓ Add attribution.txt file to give image authors credit

Icon Library - not a python package

- ✓ Download icons in 24x24 px format from flaticon.com
- ✓ Create buttons with icons or icons & text to enhance the GUI
- ✓ Add attribution.txt file to give icon authors credit

Project Setup

Language: Python 3.11

IDE: PyCharm 2023.2.1 (Community Edition)

Project directory: D:/EETools/SimpleDiagramEditor

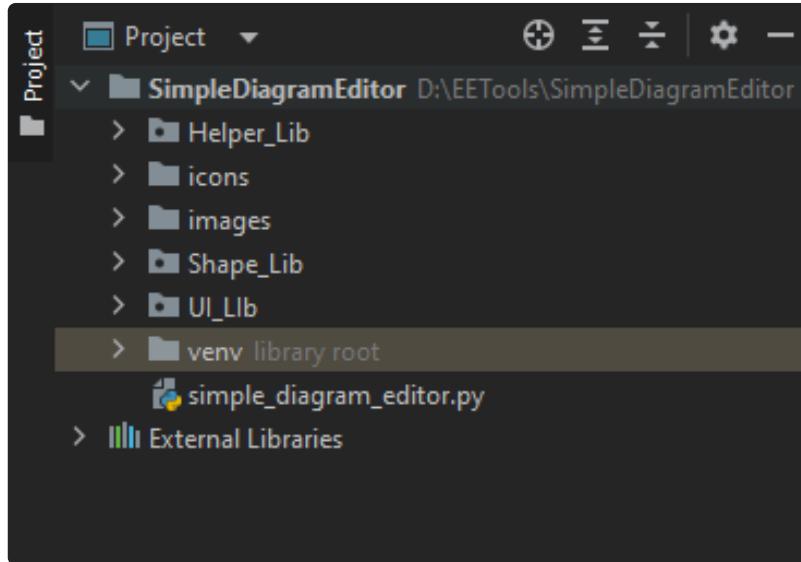
Graphics library: CustomTkinter (<https://customtkinter.tomschimansky.com/>)

External libraries:

- ✓ pip install customtkinter
- ✓ python.exe -m pip install --upgrade pip
- ✓ pip install ctkcolorpicker

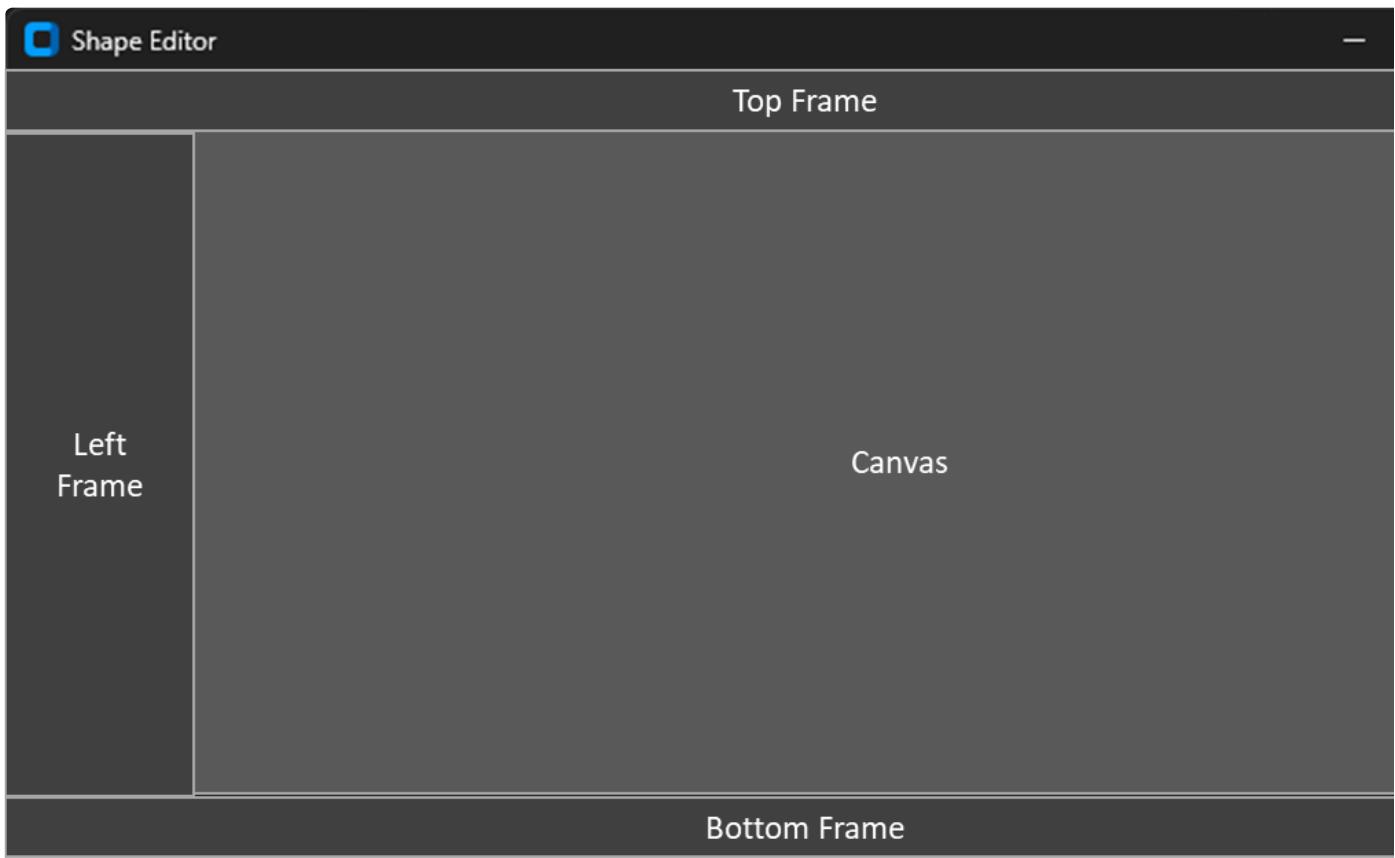
Initial Directory Structure

Per the specifications, the initial directory structure has three python packages: Helper_Lib, Shape_Lib, and UI_Lib. Recall that you must add `__init__.py` to each package directory to make it a package. We also need two standard directories: icons and images.



User Interface - Simple Diagram Editor App Class

The initial user interface design is shown in the image below. It consists of a CustomTkinter window with a canvas widget and three custom frame widgets. Custom widgets allow us to create a modular UI design so that the main file class is small and manageable.



To understand this software development process, lets create the UI in the main file and then "modularize" the code by creating custom classes in module files.

Main Application Class

Create a new file in the Simple Diagram Editor called `simple_diagram_editor.py`.

Import statements are usually found at the top of the file. Initially, we import the `CustomTkinter` library which has widgets for creating a modern user interface with light and dark modes.

```
import customtkinter as ctk
```

Next we create the Simple Diagram Editor App Class which inherits from the `ctk` window class, so we will need two initializers: one for the derived class and one for the "super" or base class.

```
class SimpleDiagramEditorApp(ctk.CTk):  
    def __init__(self):
```

```
super().__init__()
```

We can instantiate this class in the "main" entry point code as follows:

```
if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

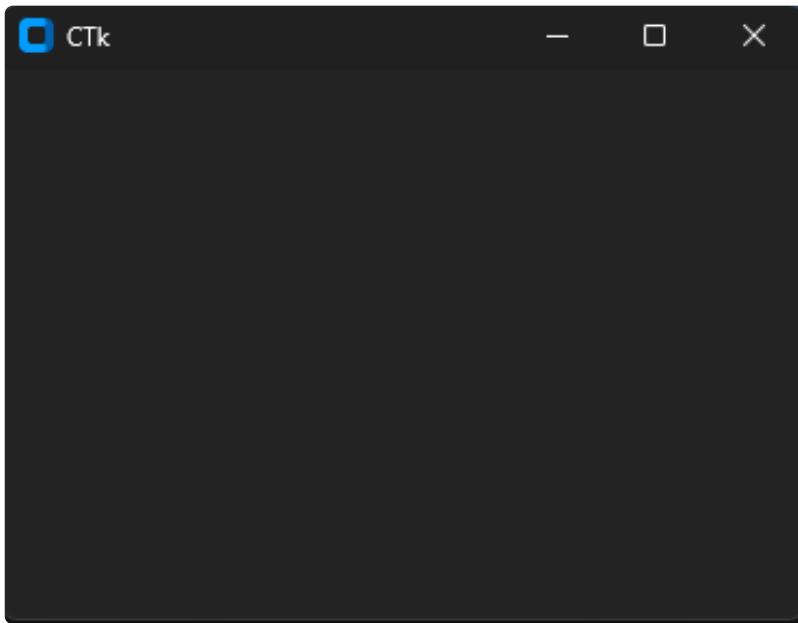
Here is the complete simple_diagram_editor.py file at this point

```
import customtkinter as ctk

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

The program is now executable and if you run it, you will see a blank CustomTkinter window.

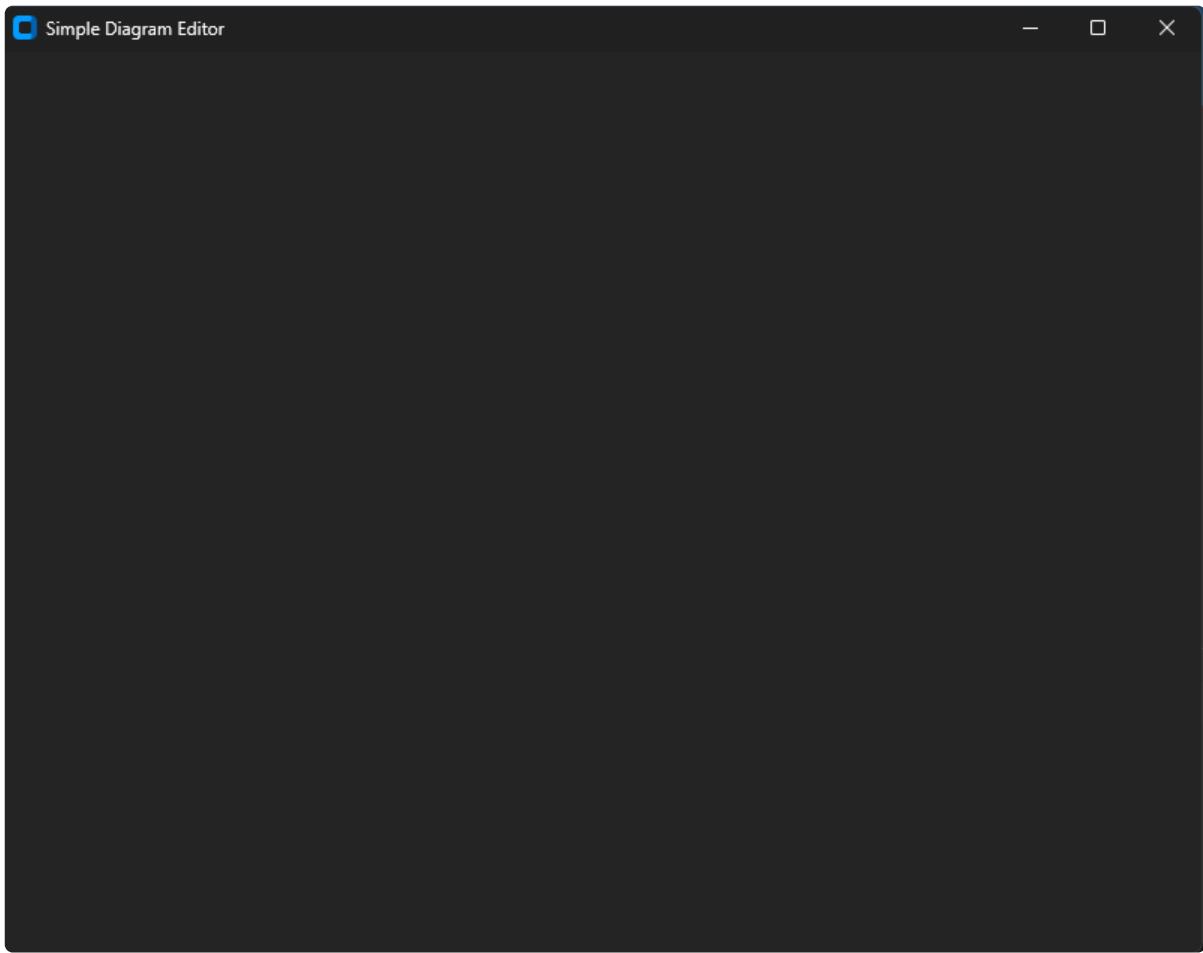


Not very exciting, but it proves that our project is setup and ready for development. Sort of a "Hello World" program with a user interface.

To improve the window, set the window geometry which defines its size and location using a weird string notation `"WxHxXxY"` where W = window width, H = window height, X = x location coordinate, and Y = y location coordinate in pixels. Set the window title to "Simple Diagram Editor".

```
...
class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")
...
...
```

A code snippet is indicated by the three dots (...). Note that I did not show the import statement or the main entry point code in this code snippet. You should assume that they are still there. Run the program again to see that the window size, location, and title are set.



Creating the User Interface

Add code that creates a canvas and three frames.

```
...
class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = ctk.CTkCanvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

...
...
```

If you run the program now, there is no change. We need to add the new widgets to the window using one of the three tkinter layout managers: place, pack, or grid. For this application we will use the pack or packer layout manager which "packs" or creates a layout relative to earlier widgets so the order that widgets are packed is important in determining the relative position and size of the widget. We want the top and bottom frames to extend from the left side of the window to the right side so we pack those widgets first. Set the `side=` option to Top for the Top Frame and Bottom for the Bottom Frame. Then we add the left frame and the canvas which will be packed between the top and bottom frame. Set the `side=` option to Left for the Left Frame and Right for the Canvas.

```
    . . .

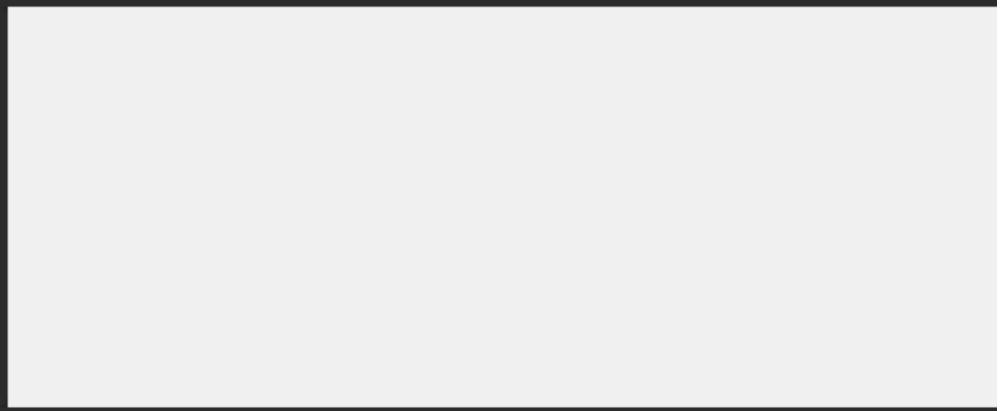
class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = ctk.CTkCanvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

    . . .
```

Run the program to see the initial user interface "skeleton". It looks strange because the frames and canvas have "autosized" to fill the available space. Don't worry or attempt to adjust the frame sizes as we will take advantage of the autosize feature when widgets are added to the frame. The canvas will autosize also to take up all available space not used by the frames.



Now add test buttons to the Top Frame and Left Frame. Also, add a label to the Bottom Frame as follows.

... . . .

```
self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

# Add top frame widgets here
button = ctk.CTkButton(self.top_frame, text="File")
button.pack(side=ctk.LEFT, padx=5, pady=5)

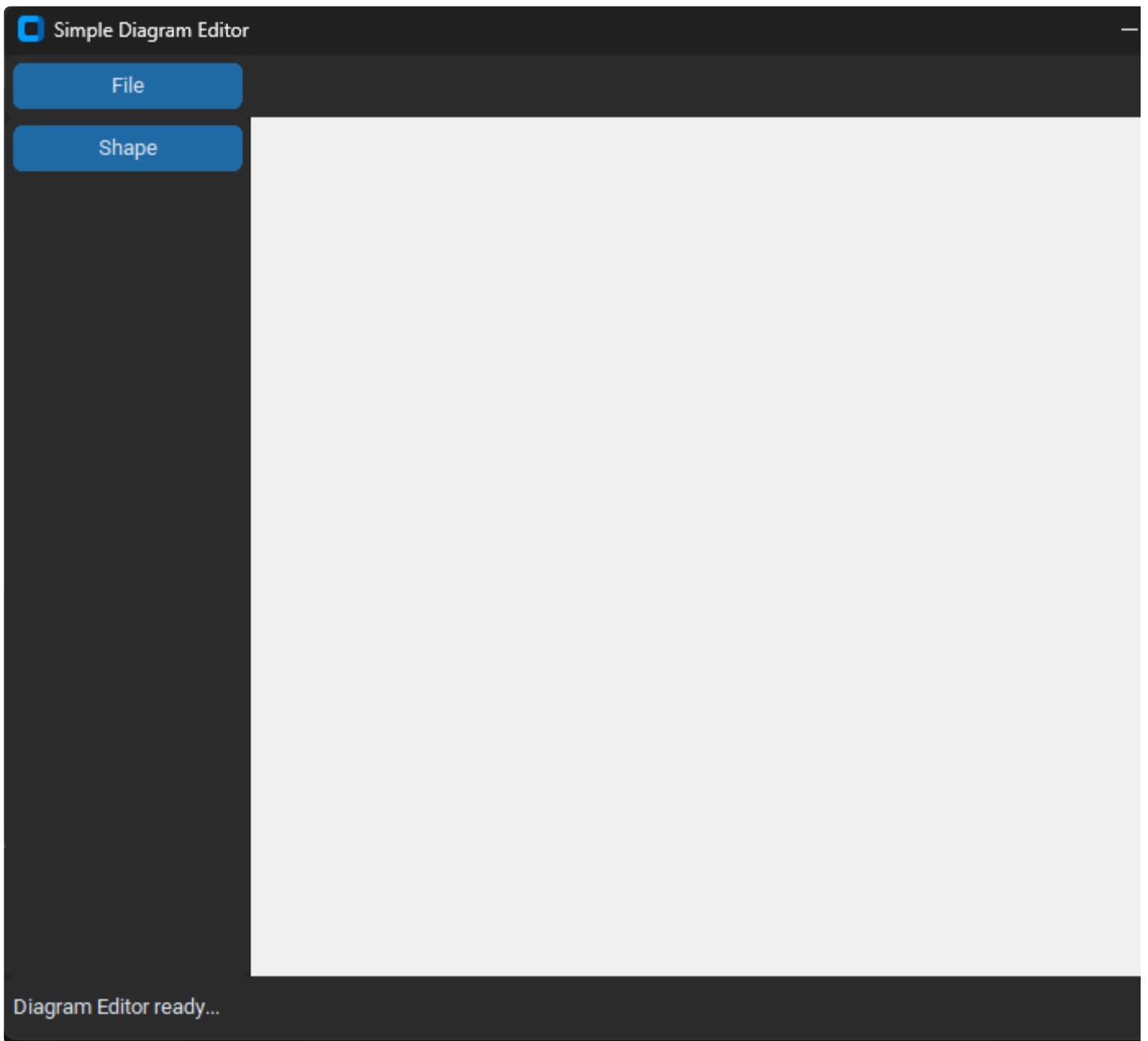
# Add left frame widgets here
button = ctk.CTkButton(self.left_frame, text="Shape")
```

```
button.pack(side=ctk.TOP, padx=5, pady=5)

# Add bottom frame widgets here
label = ctk.CTkLabel(self.bottom_frame, text="Diagram Editor ready... ")
label.pack(side=ctk.LEFT, padx=5, pady=5)

if __name__ == "__main__":
    . . .
```

Run the program now and note that the Top Frame, Left Frame, and Bottom Frame have autosized to the test widgets. Also, the canvas now fills the available space. If you maximize the window, note that all frames and the canvas resize automatically.



Here is the full program code listing for simple_diagram_editor.py

```
import customtkinter as ctk

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        # Create canvas and frames
        self.canvas = ctk.CTkCanvas(self)
```

```

    self.top_frame = ctk.CTkFrame(self)
    self.left_frame = ctk.CTkFrame(self)
    self.bottom_frame = ctk.CTkFrame(self)

        # Pack canvas and frames on the ctk window
    self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
    self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
    self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

    self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add top frame widgets here
    button = ctk.CTkButton(self.top_frame, text="File")
    button.pack(side=ctk.LEFT, padx=5, pady=5)

    # Add left frame widgets here
    button = ctk.CTkButton(self.left_frame, text="Shape")
    button.pack(side=ctk.TOP, padx=5, pady=5)

    # Add bottom frame widgets here
    label = ctk.CTkLabel(self.bottom_frame, text="Diagram Editor ready...")
    label.pack(side=ctk.LEFT, padx=5, pady=5)

if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()

```

We could keep adding widgets to the main diagram editor app but the code would quickly expand and become more difficult to read and maintain. At this point, we will modularize the code into custom class modules.

Canvas Class

In the user interface library (UI_Lib), create a new file called `canvas.py`.

```

import customtkinter as ctk

class Canvas(ctk.CTkCanvas):
    def __init__(self, master):

```

```
super().__init__(master)
```

Modify `UI_Lib/__init__.py` file to import the class

```
from UI_Lib.canvas import Canvas
```

The Canvas Class inherits from the CustomTkinter Canvas Class. To use this class we need to modify `simple_diagram_editor.py` to import the class and use it to instantiate our `self.canvas` object.

```
import customtkinter as ctk
from UI_Lib import Canvas # Added new import

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self) # Changed to use the custom Canvas class
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

    ...
    . . .
```

If you run the program, you should see no change to the user interface.

Top Frame Class

The Top Frame class will contain widgets for File Menu, Settings Menu, Help Menu, and shape appearance controls such as shape fill color, shape border color, and shape border width.

In the user interface library (`UI_Lib`), create a new file called `top_frame.py`. The Top Frame class inherits from the CustomTkinter Frame Class. The Frame Class needs a "parent" widget where the frame will be created. Add the test button here for the Top Frame and

remove it from the main diagram editor app. Note that the text button parent changes from `self.top_frame` to `self` which refers to the Top Frame Class.

```
import customtkinter as ctk

class TopFrame(ctk.CTkFrame):
    def __init__(self, parent):
        super().__init__(parent)

        # Add top frame widgets here
        button = ctk.CTkButton(self, text="File") # Changed parent to self
        button.pack(side=ctk.LEFT, padx=5, pady=5)
```

Modify `UI_Lib/__init__.py` file to import the class

```
from UI_LIB.canvas import Canvas
from UI_LIB.top_frame import TopFrame
```

To use this class we need to modify `simple_diagram_editor.py` to import the class and use it to instantiate our `self.top_frame` object.

```
import customtkinter as ctk
from UI_LIB import Canvas, TopFrame # Changed to import the TopFrame class

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self) # Changed to use the custom TopFrame
class
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)
```

```
# Deleted the top frame test button code
```

```
... . . .
```

Running the program again, you will see that there is no change.

Left Frame Class

The Left Frame Class will contain the Shape creation buttons for creation of basic shapes such as Rectangles, Ovals, Triangles (Polygons), Lines, Text, and Images (Pictures).

Similar to the Top Frame Class, create a new module In the user interface library (UI_Lib) called left_frame.py. Add the test button here for the Left Frame and remove it from the main diagram editor app.

```
import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        button = ctk.CTkButton(self.left_frame, text="Shape")
        button.pack(side=ctk.TOP, padx=5, pady=5)
```

Don't forget to modify `UI_Lib/__init__.py` file to import the class.

```
from UI_Lib.canvas import Canvas
from UI_Lib.top_frame import TopFrame
from UI_Lib.left_frame import LeftFrame
```

Modify the main file to import the Left Frame Class and use it to create the left frame object.

```

import customtkinter as ctk
from UI_Lib import Canvas, TopFrame, LeftFrame # Added import for LeftFrame


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self) # Changed to use the LeftFrame class
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Deleted test buttons for TopFrame and LeftFrame

        # Add bottom frame widgets here
        label = ctk.CTkLabel(self.bottom_frame, text="Diagram Editor ready... ")
        label.pack(side=ctk.LEFT, padx=5, pady=5)

    ...

```

Run the program and verify that there are no changes to the window.

Bottom Frame Class

The Bottom Frame Class will hold a label for user messages, modes, and status.

Create the Bottom Frame Class module called bottom_frame.py in the UI_Lib directory.
Don't forget to add the test label.

```

import customtkinter as ctk

class BottomFrame(ctk.CTkFrame):

```

```
def __init__(self, parent):
    super().__init__(parent)

    self.message = ctk.CTkLabel(self, text="Diagram editor ready...")
    self.message.pack(side=ctk.LEFT, padx=5, pady=5)
```

UI_Lib/__init__.py

```
from UI_Lib.canvas import Canvas
from UI_Lib.top_frame import TopFrame
from UI_Lib.left_frame import LeftFrame
from UI_Lib.bottom_frame import BottomFrame # Added import for ButtonFrame
```

Update main file

```
import customtkinter as ctk
from UI_Lib import Canvas, TopFrame, LeftFrame, BottomFrame # Added import for
BottomFrame

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self)
        self.bottom_frame = BottomFrame(self) # Changed to use the BottomFrame
class

        # Deleted code for test buttons and test label

    . . .
```

Again, run the program to verify that there are no changes to the simple user interface.

Rectangle Class

We could create a simple rectangle from the main application in one line of code as follows:

simple_diagram_editor.py

```
 . . .

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

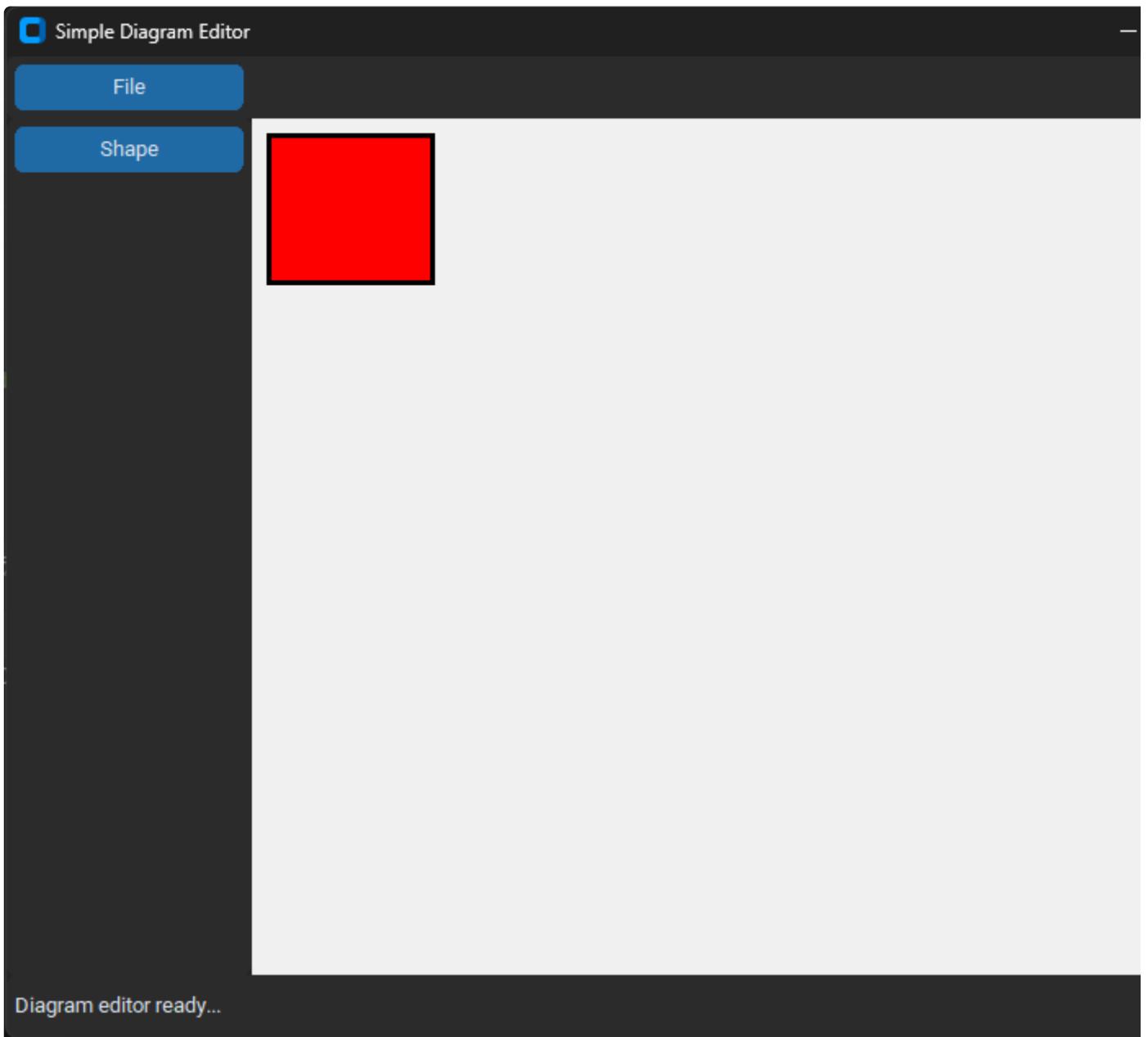
        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self)
        self.bottom_frame = BottomFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add shapes here
    rect = self.canvas.create_rectangle(10, 10, 110, 100, fill="red",
outline="black", width=3)  # Create a rectangle!

. . .
```



Cool! We created a Rectangle with fill color = "red", border (outline) color = "black", and border width = 3. We positioned the shape at the point 10, 10 which is the upper-left corner of the shape and at point 110, 110 which is the lower-right corner. Note that the width and height are 100 px.

The [Canvas](#) has shape creation methods for many shapes:

- `Rectangle (canvas.create_rectange(x1, y1, x2, y2, fill="red", outline="black", width=3))`
- `Oval (canvas.create_oval(x1, y1, x2, y2, fill="red", outline="black", width=3))`
- `Polygon (canvas.create_polygon(points, fill="red", outline="black", width=3))`
where points is a list of points

- Arc (`canvas.create_arc(x1, y1, x2, y2, fill="red", outline="black", width=3)`)
- Bitmap (`canvas.create_bitmap(x, y, bitmap=filename, anchor=ctk.CENTER)`)
- Image (`canvas.create_image(x, y, image=filename, anchor=ctk.CENTER)`)
- Line (`canvas.create_line(x1, y1, x2, y2, fill="red", width=3)`)
- Text (`canvas.create_text(x, y, text="", anchor=ctk.CENTER)`)
- Window(`canvas.create_window(x, y, window=w, anchor=ctk.CENTER)`)

Canvas also has [methods](#) for modifying a shape drawn on it. Here are some methods we will use:

- `canvas.bbox(tagOrId)` - Returns a tuple (x1, y1, x2, y2) describing a rectangle that encloses all the objects specified by `tagOrId`. We will need this to get the dimensions and location for Text and Image shapes.
- `canvas.coords(tagOrId, points)` - Use to move a shape by passing a new set of points or coordinates.
- `canvas.delete(tagOrId)` - Deletes a shape from the canvas
- `canvas.itemconfigure(tagOrId, option)` - Use to change a shape color or width.

Rectangle Class

The Rectangle Class wraps the `canvas.create_rectangle()` method in a class structure so we can take control of the rectangle shape in an object-oriented way.

In the Shape Library (Shape_Lib) create a new file called `rectangle.py`. Name the class `Rectangle`. In the class initializer we need a reference to the canvas for the `self.canvas.create_rectangle()` method. Create attributes for the rectangle coordinates and set default values for fill color, border color, and border width.

`rectangle.py`

```
class Rect:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
```

```
    self.border_color = "black"
    self.border_width = 3
```

Next, add a place to store the rectangle canvas id which is an integer index to the canvas shape list. Add a create() method that creates the rectangle and sets the shape id.

```
class Rect:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None

    def create(self):
        """Create the shape once!
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)
```

In `Shape_Lib/__init__.py` import the Rectangle Class

```
from Shape_Lib.rectangle import Rectangle
```

This is enough code to create a rectangle. In the main application file, import the Rectangle class and replace the code that creates a rectangle with code that instantiates a rectangle object and calls the object create() method.

```
import customtkinter as ctk
from UI_Lib import Canvas, TopFrame, LeftFrame, BottomFrame
from Shape_Lib import Rectangle # Add import for Rectangle
```

```
class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

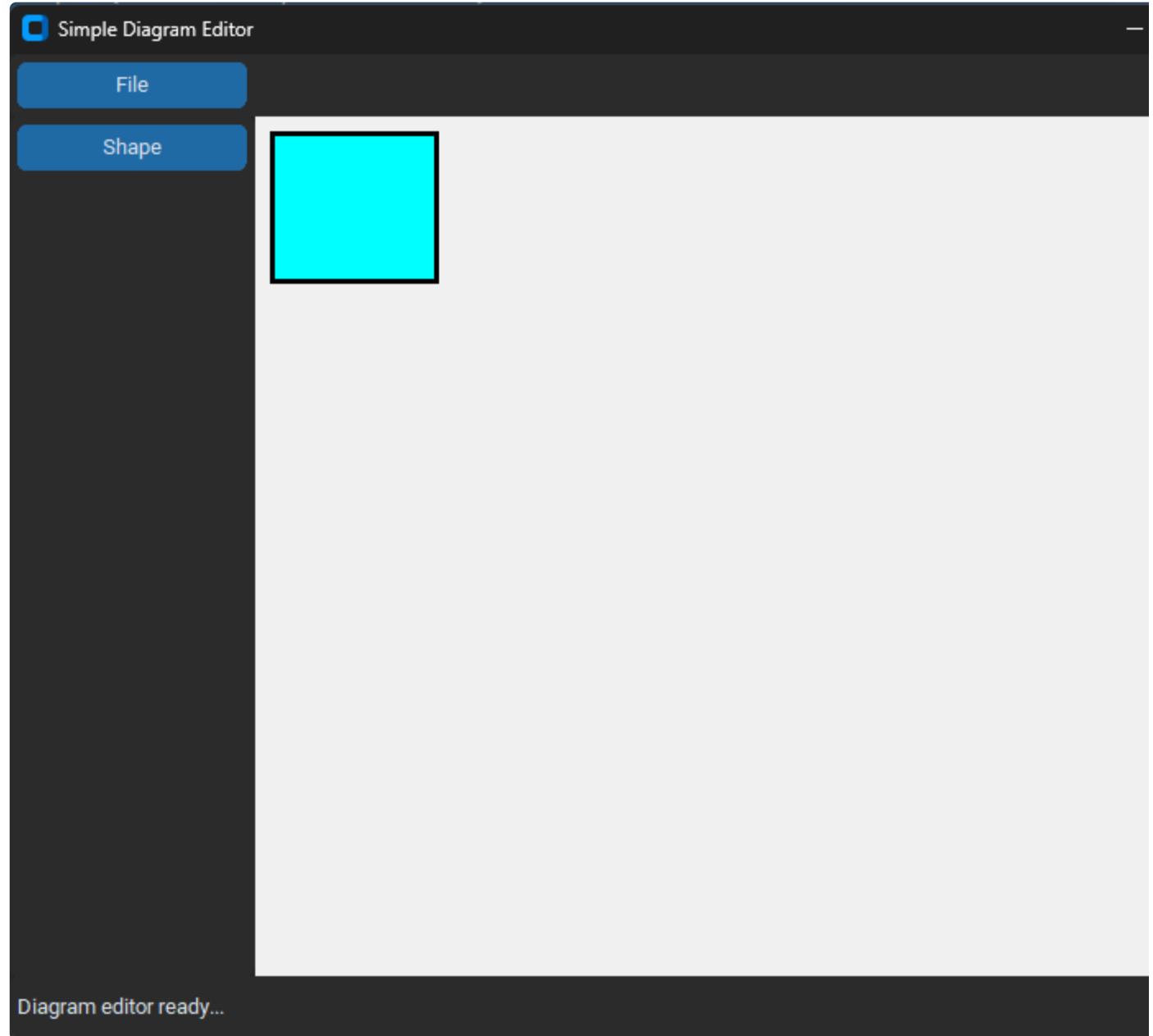
        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self)
        self.bottom_frame = BottomFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Add shapes here
        # rect = self.canvas.create_rectangle(10, 10, 110, 100, fill="red",
        outline="black", width=3) # Commented out old code
        rect = Rectangle(self.canvas, 10, 10, 110, 100) # Added code to use
        the Rectangle Class
        rect.create() # Call the Rectangle create() method
```

We now have a cyan rectangle drawn from the Rectangle Class.



Next add an update() method to the Rectangle Class

```
class Rectangle:  
    def __init__(self, canvas, x1, y1, x2, y2):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.fill_color = "cyan"  
        self.border_color = "black"  
        self.border_width = 3
```

```
self.id = None

def create(self):
    """Create the shape once!
    self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                            fill=self.fill_color,
                                            outline=self.border_color,
                                            width=self.border_width)

def update(self):
    """Update the shape, don't recreate it!
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, outline=self.border_color)
    self.canvas.itemconfig(self.id, width=self.border_width)
```

We can test our classes locally before importing them into the main application. Lets add a `__repr__()` method that creates a custom string for the `print()` methods. Also, we can add a main section to test the class.

```
def update(self):
    """Update the shape, don't recreate it!"""
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, outline=self.border_color)
    self.canvas.itemconfig(self.id, width=self.border_width)

def __repr__(self):
    return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

if __name__ == "__main__":
    import customtkinter as ctk

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Rectangle Class")

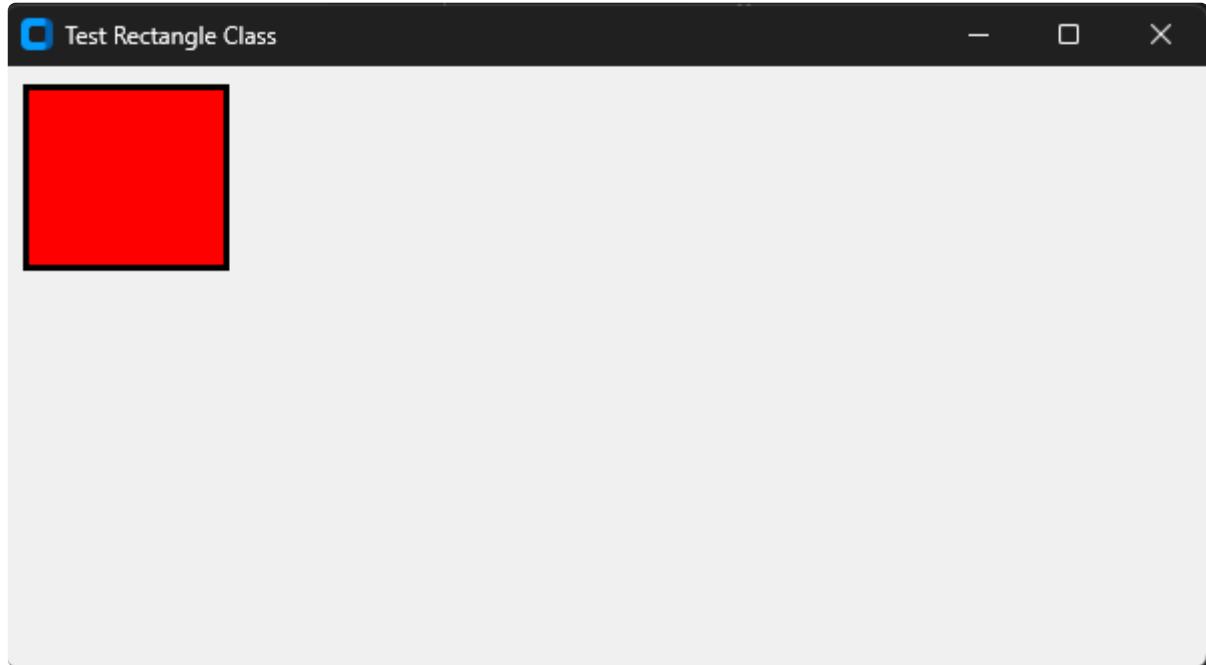
    canvas = ctk.CTkCanvas()
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    rect = Rectangle(canvas, 10, 10, 110, 100)
    rect.create()
    print(rect)

    # Change the fill color to test the update() method
    rect.fill_color = "red"
    rect.update()

app.mainloop()
```

Run the Rectangle Class to see the test window.



Console Output

```
Rectangle: x1, y1, x2, y2 = (10, 10, 100, 100)
fill color: cyan
border_color: black
border_width: 3
```

Ok, all Rectangle Class tests passed so lets add the class to the main application.

Now lets create the rectangle from the Left Frame shape menu, we can modify the test button to perform this task.

In UI_Lib/left_frame.py, import the Rectangle Class

```
import customtkinter as ctk
from Shape_Lib.rectangle import Rectangle
```

In the class initializer, we will need a reference to the application canvas

```
class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas): # Added canvas to the initializer
```

```
super().__init__(parent)
self.canvas = canvas # Create a local canvas attribute

# Add left frame widgets here
rect_button = ctk.CTkButton(self, text="Rectangle",
command=self.create_rect) # Change to Rectangle button with command
rect_button.pack(side=ctk.TOP, padx=5, pady=5) # Change to Rectangle
button
```

Add the `create_rect()` method to handle creation of the rectangle

```
def create_rect(self):
    shape = Rectangle(self.canvas, 10, 10, 110, 110)
    shape.create()
```

Here is the complete Left Frame class definition

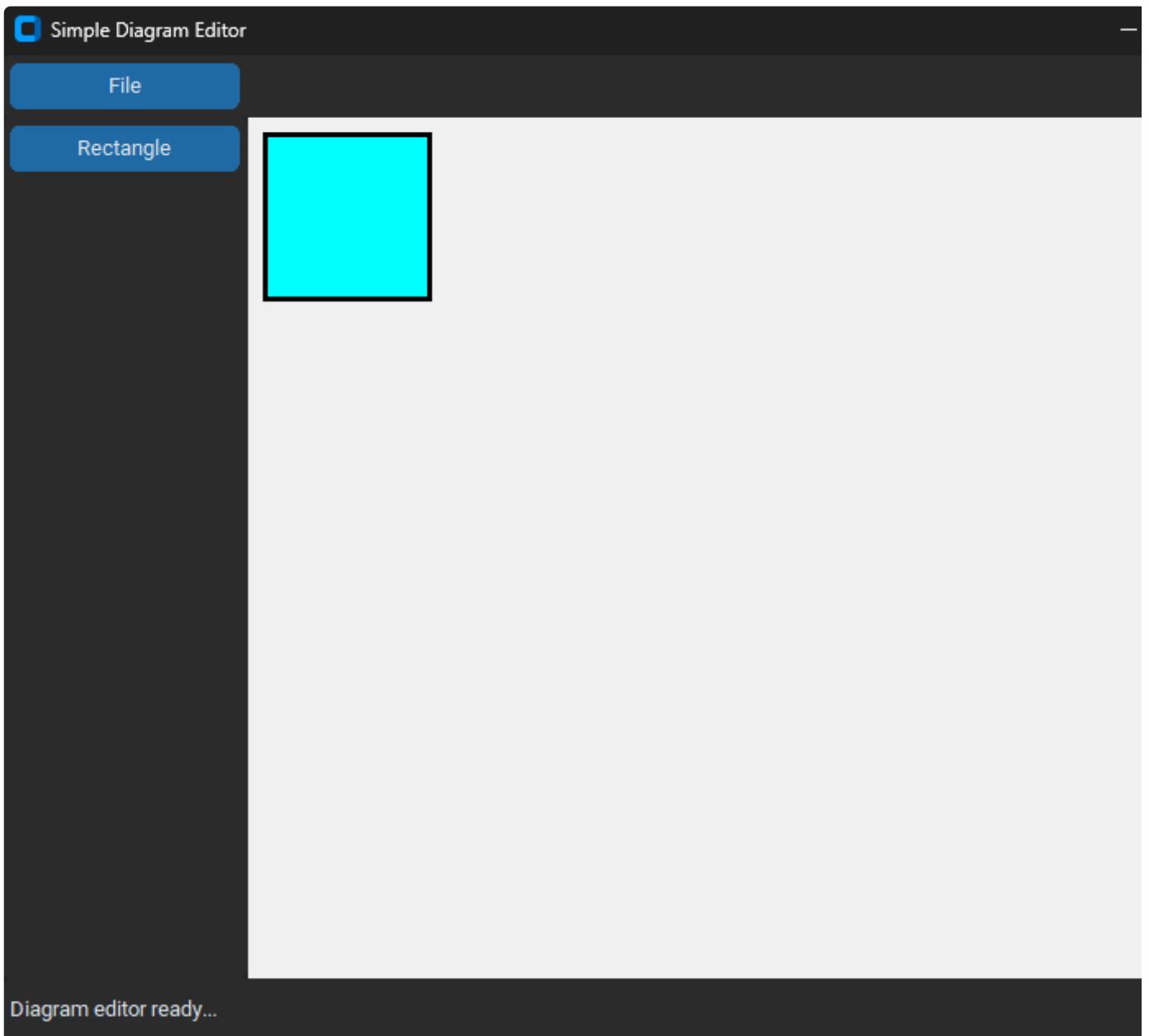
```
import customtkinter as ctk
from Shape_Lib.rectangle import Rectangle

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self, text="Rectangle",
command=self.create_rect)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

    # Left frame button handlers
    def create_rect(self):
        shape = Rectangle(self.canvas, 10, 10, 110, 110)
        shape.create()
        self.canvas.shape_list.append(shape)
        self.canvas.mouse.selected_shape = shape
```

Run the program, click the Rectangle button and confirm that a cyan rectangle appears on the canvas.



Now that the rectangle shape can be drawn on the canvas from the Left Frame menu, lets add the following capabilities:

- Mouse move
- Keyboard rotate
- Set fill color
- Set border color
- Set border width
- Mouse draw
- Mouse resize

Point Class

Lets enhance the [Point Class](#) created in the Line Editor Application to add a `__repr__()` method and a main test block to test the class.

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __repr__(self):  
        return "Point: (" + str(self.x) + ", " + str(self.y) + ")"  
  
  
if __name__ == "__main__":  
    a_point = Point(10, 15)  
    print(a_point)  
    print("a_point.x = ", a_point.x)  
    print("a_point.y = ", a_point.y)
```

Console Output after running the program.

```
Point: (10, 15)  
a_point.x = 10  
a_point.y = 15
```

That was pretty easy and we can use this for debug during development.

Move Shape with Mouse

All mouse events will be handled by the Mouse Class.

- Import the Point Class
- Initialize the Mouse Class with a reference to the canvas
- Add an attribute to keep track of the currently selected shape
- Create two offset points and initialize
- Create bindings for left mouse down, drag, and up

- Create event handlers for left mouse down, draw, and up to move the selected shape
- Add a hit test to determine if a shape is selected by left mouse down

mouse.py

```
from Helper_Lib.point import Point

class Mouse:
    def __init__(self, canvas):
        """Class to manage mouse events"""
        self.canvas = canvas
        self.selected_shape = None

        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

    # Mouse bindings
    self.canvas.bind("<Button-1>", self.move_left_down)
    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)
        if self.selected_shape:
            # print("Shape found: ", self.selected_shape)
            x1, y1 = self.selected_shape.x1, self.selected_shape.y1
            x2, y2 = self.selected_shape.x2, self.selected_shape.y2
            self.offset1.x = x - x1
            self.offset1.y = y - y1
            self.offset2.x = x - x2
            self.offset2.y = y - y2

    def move_left_drag(self, event):
        if self.selected_shape:
            x = event.x - self.offset1.x
            y = event.y - self.offset1.y
            self.selected_shape.x1, self.selected_shape.y1 = x, y
            x = event.x - self.offset2.x
            y = event.y - self.offset2.y
            self.selected_shape.x2, self.selected_shape.y2 = x, y
            self.canvas.redraw_shapes()
```

```

def move_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0

def select_hit_test(self, x, y):
    for s in self.canvas.shape_list:
        if s.x1 <= x <= s.x2 and s.y1 <= y <= s.y2:
            self.selected_shape = s

```

We can add a main test section to test the mouse before it is integrated into the main application.

```

. . .

if __name__ == "__main__":
    import customtkinter as ctk
    from Shape_Lib import Rectangle
    from UI_Lib import Mouse, Canvas

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Mouse Class")

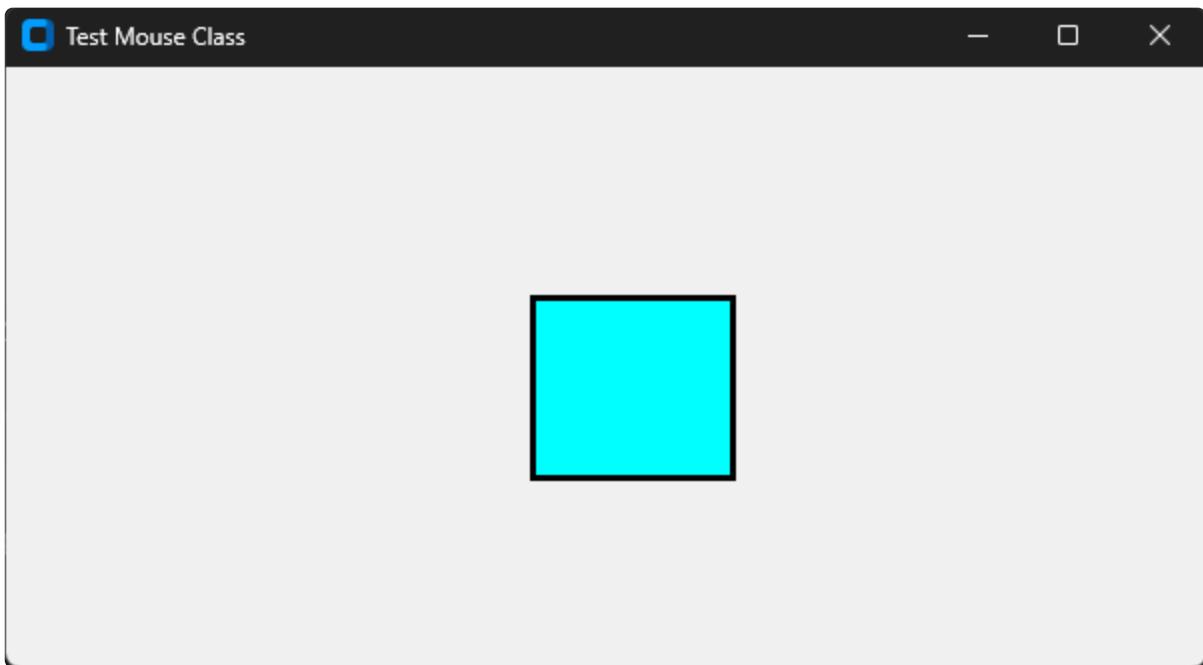
    canvas = Canvas(app)
    mouse = Mouse(canvas)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    rect = Rectangle(canvas, 10, 10, 110, 100)
    rect.create()
    canvas.shape_list.append(rect)

    app.mainloop()

```

Run the program and verify that a cyan rectangle is created and that you can move the rectangle using the left mouse button.



Canvas Class Modifications

The Canvas Class modifications are:

- Add a shape list
- Create an instance of the Mouse Class
- Add a redraw shapes method that iterates over the shape list and calls the shape update() method

canvas.py

```
import customtkinter as ctk
from UI_Lib.mouse import Mouse

class Canvas(ctk.CTkCanvas):
    def __init__(self, master):
        super().__init__(master)
        self.shape_list = []
        self.mouse = Mouse(self)

    def redraw_shapes(self):
        for s in self.shape_list:
            s.update()
```

We can add a main test section to test the canvas class before it is integrated into the main application.

```
...
if __name__ == "__main__":
    import customtkinter as ctk
    from Shape_Lib import Rectangle
    from UI_Lib import Mouse, Canvas

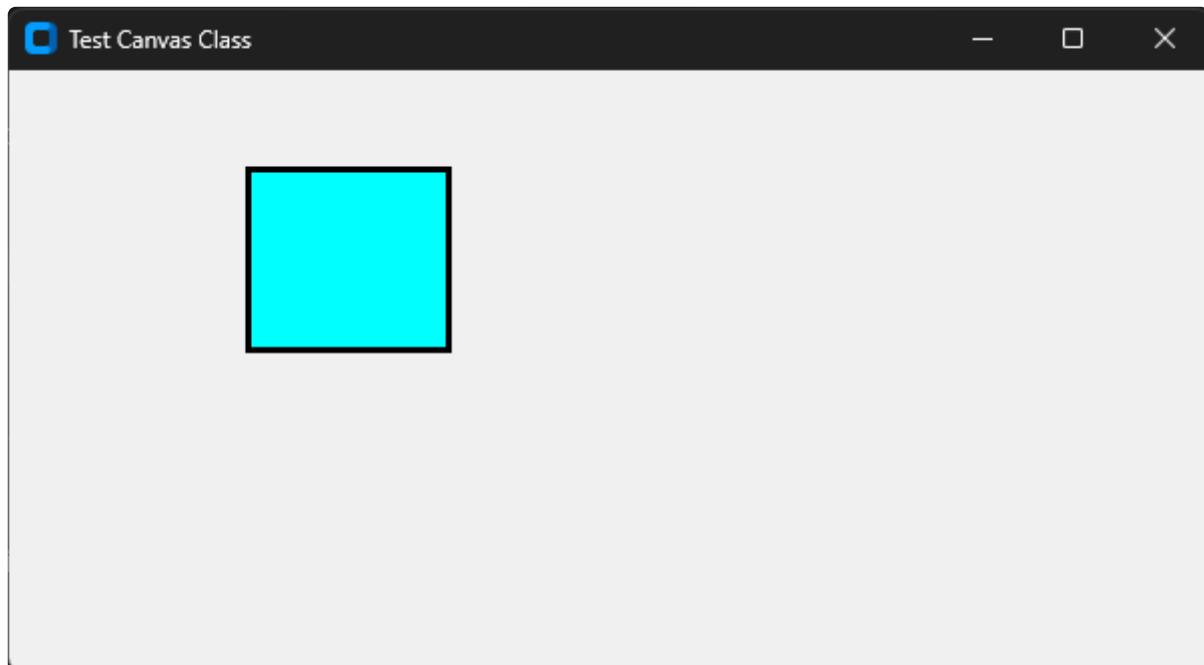
    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Mouse Class")

    canvas = Canvas(app)
    mouse = Mouse(canvas)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    rect = Rectangle(canvas, 10, 10, 110, 100)
    rect.create()
    canvas.shape_list.append(rect)

    app.mainloop()
```

Run the program and confirm that a cyan rectangle is created and that you can move it with the left mouse button.



Shape Selection

We need a visual indicator that a shape is selected. We will use a red rectangle that is 5 px larger than the enclosed shape as a simple selector, for now.

Selection rules:

- Select a shape by left click on the shape
- Unselect all shapes by left click on the canvas, not on a shape
- Select more than one shapes by clicking on each shape

In the Rectangle Class, add a new attribute called `self.is_selected` which is a boolean variable and set it to False. Also, add a `self.sel_id` attribute and set it to None. The `sel_id` will store the index of the selection rectangle so it can be updated in the `update()` method.
rectangle.py

```
class Rectangle:  
    def __init__(self, a_canvas, x1, y1, x2, y2):  
        self.canvas = a_canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
        self.fill_color = "cyan"  
        self.border_color = "black"  
        self.border_width = 3  
  
        self.id = None  
        self.sel_id = None # Added  
        self.is_selected = False # Added  
  
    . . .
```

In the rectangle `create()` method create a red rectangle with coordinates 5 px larger than the original rectangle with transparent fill, outline color of red, and width of 2 px. Store the selector id in `self.sel_id`. Change the state of the selector rectangle to 'hidden' using `canvas.itemconfig()`.

```

. . .

def create(self):
    """Create the shape once!
    self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                            fill=self.fill_color,
                                            outline=self.border_color,
                                            width=self.border_width)

    self.sel_id = self.canvas.create_rectangle(self.x1-5, self.y1-5,
self.x2+5, self.y2+5, fill=None, # Added
                                                outline="red", width=2)
    self.canvas.itemconfig(self.sel_id, state='hidden') # Added

. . .

```

Modify the rectangle update() method to update the `coords` of the selection rectangle and change the state to 'normal' if selected or 'hidden' if not selected.

```

. . .

def update(self):
    """Update the shape, don't recreate it!
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, outline=self.border_color)
    self.canvas.itemconfig(self.id, width=self.border_width)

    # Update the selector position
    self.canvas.coords(self.sel_id, self.x1 - 5, self.y1 - 5, self.x2 + 5,
self.y2 + 5)
    if self.is_selected:
        self.canvas.itemconfig(self.sel_id, state='normal')
    else:
        self.canvas.itemconfig(self.sel_id, state='hidden')

. . .

```

In the Mouse Class, modify the `select_hit_test()` method to set the `selected` variable to True and redraw the shapes then return to the calling method. If the for loop does not return a shape, then no shape is selected (i.e. the user clicked on the

empty portion of the canvas) and we set the selected shape to none and call the unselect all method. Add the unselect all method which iterates over the shape list and sets each shape `is_selected` variable to False.

```
 . . .

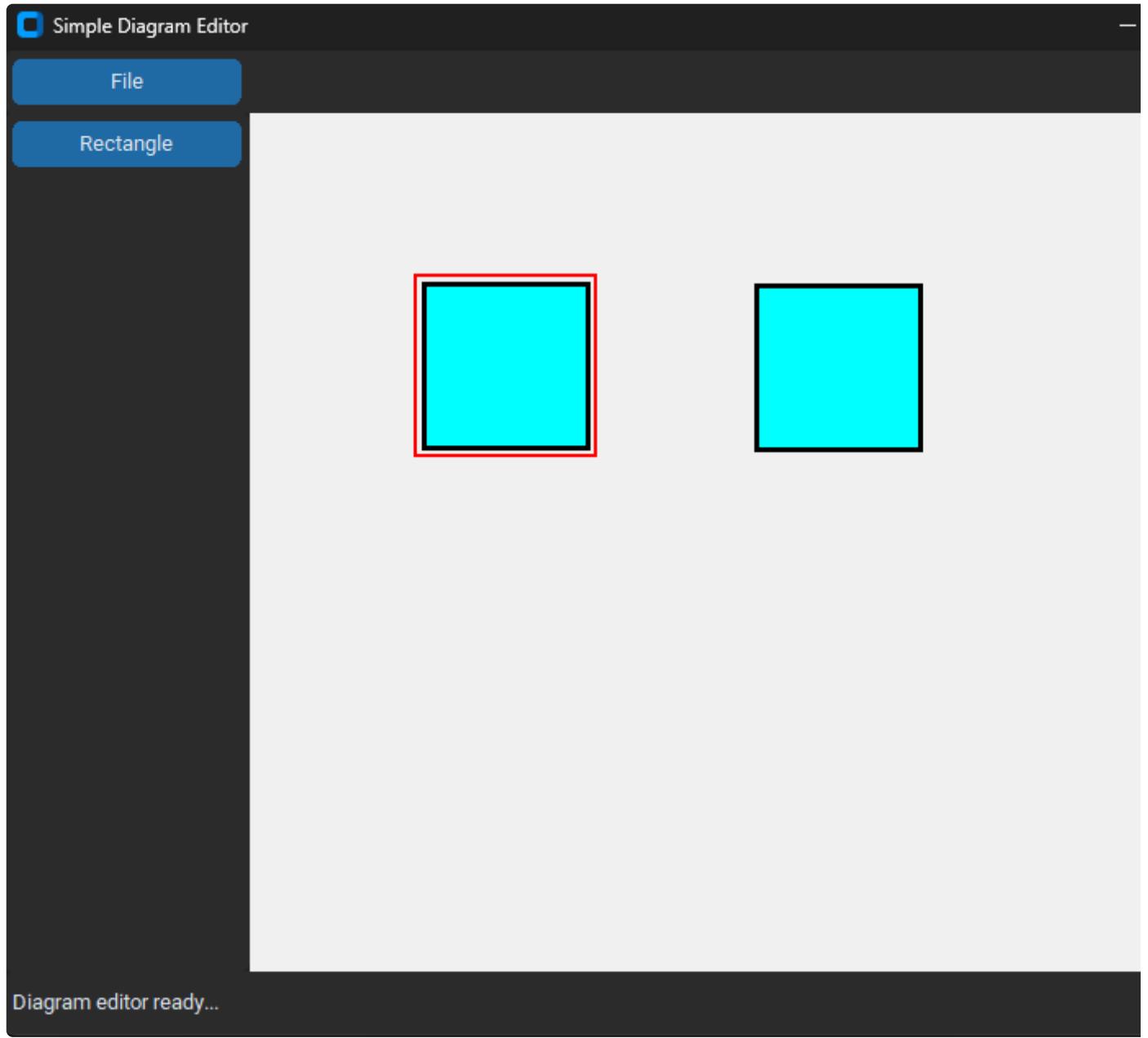
def select_hit_test(self, x, y):
    for s in self.canvas.shape_list:
        if s.x1 <= x <= s.x2 and s.y1 <= y <= s.y2:
            self.selected_shape = s
            s.is_selected = True # Added
            self.canvas.redraw_shapes() # Added
    return # Added

# No shape hit - unselect all
self.selected_shape = None # Added
self.unselect_all() # Added

def unselect_all(self): # Added new method
# print("Unselect all")
for s in self.canvas.shape_list:
    s.is_selected = False
    self.canvas.redraw_shapes()

. . .
```

Run the program and check that all selection rules defined above are satisfied.



Shape Appearance Controls

Next we will add the shape appearance controls that control the selected shape's fill color, border color, and/or border width.

- Install `tktooltip` in a terminal type `pip install tkinter-tooltip`
- Create a custom subframe called Shape Appearance Frame in `shape_appearance_frame.py`
 - Create a custom frame for Fill Color containing Button and Color Label
 - Create a custom frame for Border Color containing Button and Color Label

- Create a custom frame for Border Width containing Button with icon and width option menu
- Create tooltips for each of the controls
- Do not add it to UI_Lib/ __init__.py to avoid circular import with Top Frame Class
- Add the subframe to the Top Frame

shape_appearance_frame.py

```
import customtkinter as ctk
from CTkColorPicker import *
from tktooltip import ToolTip
from PIL import Image

class ShapeAppearanceFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        self.fill_label = None
        self.border_label = None

        self.init_fill_color_control(self)
        self.init_border_color_control(self)
        self.init_border_width_control(self)

    def init_fill_color_control(self, shape_appearance_frame):
        # Fill color frame
        fill_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
        fill_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add fill color picker
        def ask_fill_color():
            pick_color = AskColor() # open the color picker
            color = pick_color.get() # get the color string
            self.fill_label.configure(fg_color=color)
            for item in self.canvas.shape_list:
                if item.is_selected:
                    self.canvas.mouse.selected_shape.fill_color = color
                    self.canvas.redraw_shapes()

        fill_color_button = ctk.CTkButton(fill_frame, text="Fill",
text_color="black", width=50,
```

```

        command=ask_fill_color)
fill_color_button.pack(side=ctk.LEFT, padx=5, pady=5)
self.fill_label = ctk.CTkLabel(fill_frame, text="", width=30,
height=30, bg_color="white")
self.fill_label.pack(side=ctk.LEFT, padx=5, pady=5)

ToolTip(fill_color_button, msg="Fill color")

def init_border_color_control(self, shape_appearance_frame):
# Border color frame
border_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
border_frame.pack(side=ctk.LEFT, padx=5, pady=5)

# Add border (outline) color picker
def ask_border_color():
    pick_color = AskColor() # open the color picker
    color = pick_color.get() # get the color string
    self.border_label.configure(fg_color=color)
    for item in self.canvas.shape_list:
        if item.is_selected:
            self.canvas.mouse.selected_shape.border_color = color
            self.canvas.redraw_shapes()

border_color_button = ctk.CTkButton(border_frame, text="Border",
text_color="black", width=50,
command=ask_border_color)
border_color_button.pack(side=ctk.LEFT, padx=10, pady=5)
self.border_label = ctk.CTkLabel(border_frame, text="", width=30,
height=30, bg_color="white")
self.border_label.pack(side=ctk.LEFT, padx=5, pady=5)

ToolTip(border_color_button, msg="Border color")

def init_border_width_control(self, shape_appearance_frame):
# Border width frame
width_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
width_frame.configure(fg_color=("gray28", "gray28")) # set frame color
width_frame.pack(side=ctk.LEFT, padx=5, pady=5)
my_image = ctk.CTkImage(light_image=Image.open
                        ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                        dark_image=Image.open
                        ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                        size=(24, 24))

```

```

        image_label = ctk.CTkLabel(width_frame, image=my_image, text="",
corner_radius=10)
        image_label.pack(side=ctk.LEFT)

        # Add OptionMenu to top frame
def option_menu_callback(choice):
    for item in self.canvas.shape_list:
        if item.is_selected:
            self.canvas.mouse.selected_shape.border_width = choice
            self.canvas.redraw_shapes()

    option_menu = ctk.CTkOptionMenu(width_frame, values=[ "1", "2", "3",
"4", "5", "6", "7", "8", "9", "10"], width=32, command=option_menu_callback)
    option_menu.pack(side=ctk.LEFT)
    option_menu.set("3")

    ToolTip(option_menu, msg="Border width")

```

Top Frame Class

```

import customtkinter as ctk
from UI_Lib.shape_appearance_frame import ShapeAppearanceFrame # Added

class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, a_canvas): # Changed
        super().__init__(parent)
        self.parent = parent # Added
        self.canvas = a_canvas # Added

        # Add Top Frame widget here
        shape_frame = ShapeAppearanceFrame(self, self.canvas) # Added
        shape_frame.pack(side=ctk.LEFT, padx=5, pady=5) # Added

```

Notes:

- We need to use a more verbose import since both classes are in the same directory (package). We just need to specify the `directory.file_name` to import.

Main Simple Diagram Editor Application Class

```
import customtkinter as ctk
from UI_Lib import Canvas, TopFrame, LeftFrame, BottomFrame

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

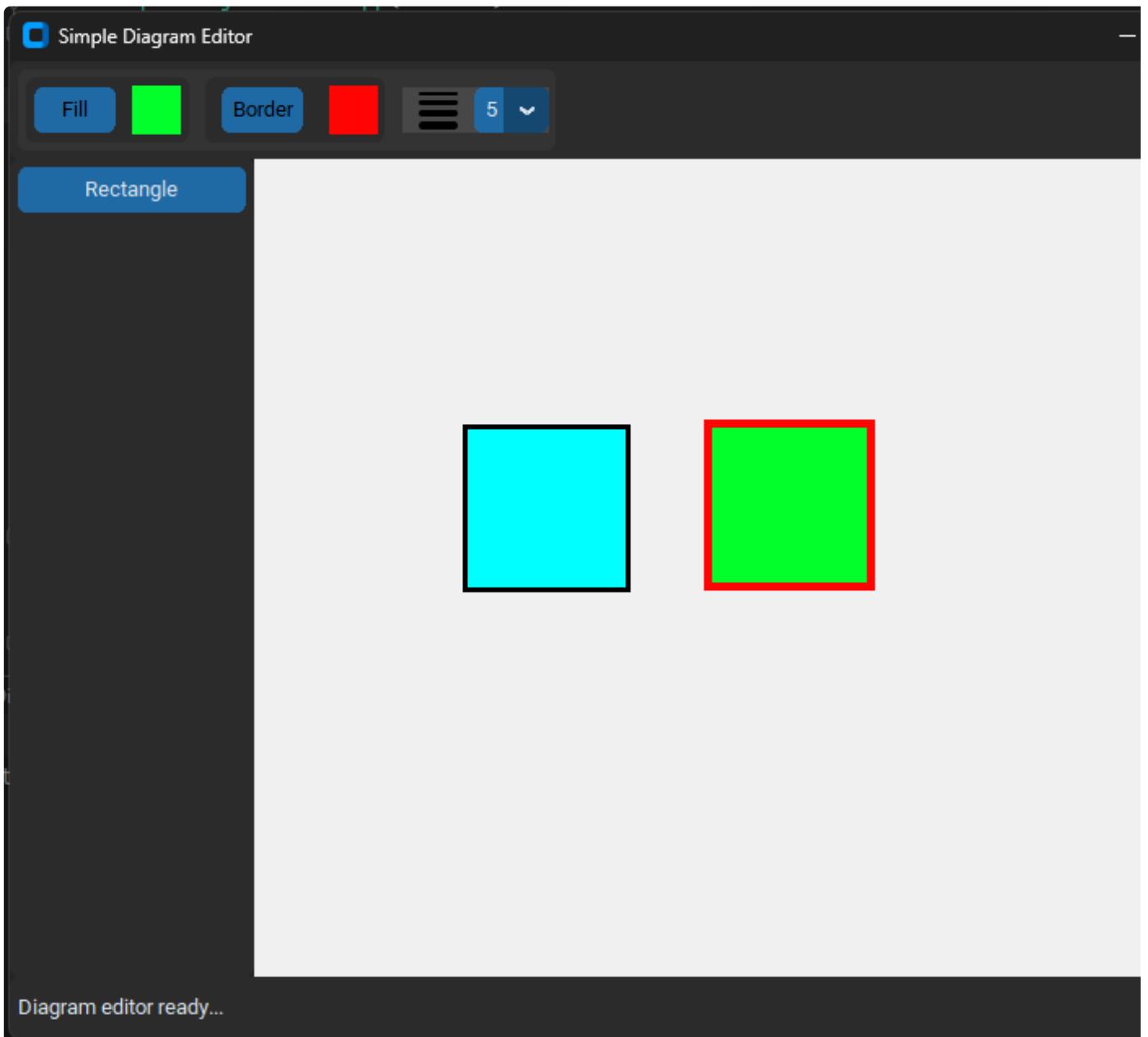
        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self, self.canvas) # Changed
        self.left_frame = LeftFrame(self, self.canvas)
        self.bottom_frame = BottomFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

Run the program, create a rectangle, select the rectangle, change the fill color, border color, and border width. Try creating two rectangles, selecting both rectangle, and change the appearance of both rectangles simultaneously! It did not work, why not?



Instead of changing only the currently selected object, iterate over all shapes and change the appearance of all selected shapes using the `canvas.itemconfig()` method. This change can be made in the Shape Appearance Frame Class.

shape_appearance_frame.py

```
import customtkinter as ctk
from CTkColorPicker import *
from tktooltip import ToolTip
from PIL import Image

class ShapeAppearanceFrame(ctk.CTkFrame):
```

```

def __init__(self, parent, canvas):
    super().__init__(parent)
    self.parent = parent
    self.canvas = canvas

    self.fill_label = None
    self.border_label = None

    self.init_fill_color_control(self)
    self.init_border_color_control(self)
    self.init_border_width_control(self)

def init_fill_color_control(self, shape_appearance_frame):
    # Fill color frame
    fill_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
    fill_frame.pack(side=ctk.LEFT, padx=5, pady=5)

    # Add fill color picker
    def ask_fill_color():
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string
        self.fill_label.configure(fg_color=color)
        for item in self.canvas.shape_list:
            if item.is_selected:
                item.fill_color = color
                self.canvas.itemconfig(item.id, fill=color) # Change

    fill_color_button = ctk.CTkButton(fill_frame, text="Fill",
text_color="black", width=50,
                                         command=ask_fill_color)
    fill_color_button.pack(side=ctk.LEFT, padx=5, pady=5)
    self.fill_label = ctk.CTkLabel(fill_frame, text="", width=30,
height=30, bg_color="white")
    self.fill_label.pack(side=ctk.LEFT, padx=5, pady=5)

    ToolTip(fill_color_button, msg="Fill color")

def init_border_color_control(self, shape_appearance_frame):
    # Border color frame
    border_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
    border_frame.pack(side=ctk.LEFT, padx=5, pady=5)

    # Add border (outline) color picker
    def ask_border_color():
        pick_color = AskColor() # open the color picker
        color = pick_color.get() # get the color string

```

```

        self.border_label.configure(fg_color=color)
        for item in self.canvas.shape_list:
            if item.is_selected:
                item.border_color = color
                self.canvas.itemconfig(item.id, outline=color) # Change

    border_color_button = ctk.CTkButton(border_frame, text="Border",
text_color="black", width=50,
                                         command=ask_border_color)
    border_color_button.pack(side=ctk.LEFT, padx=10, pady=5)
    self.border_label = ctk.CTkLabel(border_frame, text="", width=30,
height=30, bg_color="white")
    self.border_label.pack(side=ctk.LEFT, padx=5, pady=5)

    ToolTip(border_color_button, msg="Border color")

def init_border_width_control(self, shape_appearance_frame):
    # Border width frame
    width_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
    width_frame.configure(fg_color=("gray28", "gray28")) # set frame color
    width_frame.pack(side=ctk.LEFT, padx=5, pady=5)
    my_image = ctk.CTkImage(light_image=Image.open
                           ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                           dark_image=Image.open
                           ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                           size=(24, 24))

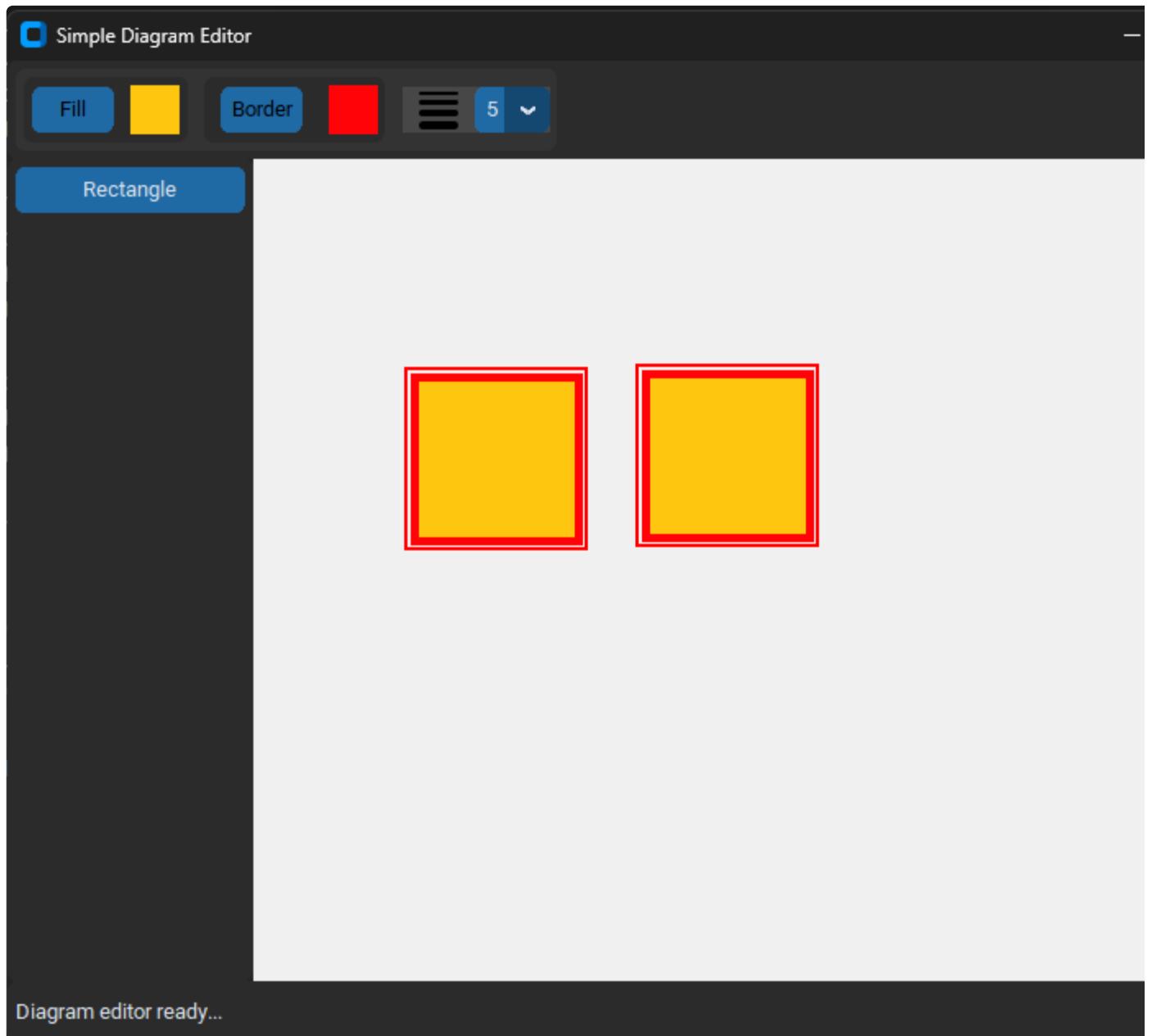
    image_label = ctk.CTkLabel(width_frame, image=my_image, text="",
corner_radius=10)
    image_label.pack(side=ctk.LEFT)

    # Add OptionMenu to top frame
    def option_menu_callback(choice):
        for item in self.canvas.shape_list:
            if item.is_selected:
                item.border_width = choice
                self.canvas.itemconfig(item.id, width=choice) # Change

    option_menu = ctk.CTkOptionMenu(width_frame, values=["1", "2", "3",
"4", "5", "6", "7", "8", "9", "10"],
                                   width=32, command=option_menu_callback)
    option_menu.pack(side=ctk.LEFT)
    option_menu.set("3")

```

```
ToolTip(option_menu, msg="Border width")
```



That works for multiple selections.

Rotate Shape with Keyboard

Add a new Keyboard Class to the UI_Lib package. The selected shape can be rotated 90 degrees by pressing the 'r' key on the keyboard.

keyboard.py

```

class Keyboard:
    def __init__(self, parent, canvas):
        """Class to manage keyboard events"""
        self.parent = parent
        self.canvas = canvas
        self.selected_shape = None

        # Declare keyboard bindings
        self.parent.bind('<r>', self.rotate_shape)

    def rotate_shape(self, _event):
        for s in self.canvas.shape_list:
            if s.is_selected:
                s.rotate()
        self.canvas.redraw_shapes()

```

We need to add a rotate() method to the rectangle class.

rectangle.py modifications

```

. . .

def rotate(self): # Added new method
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def __repr__(self):
    return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
str(self.y2) + ")\\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

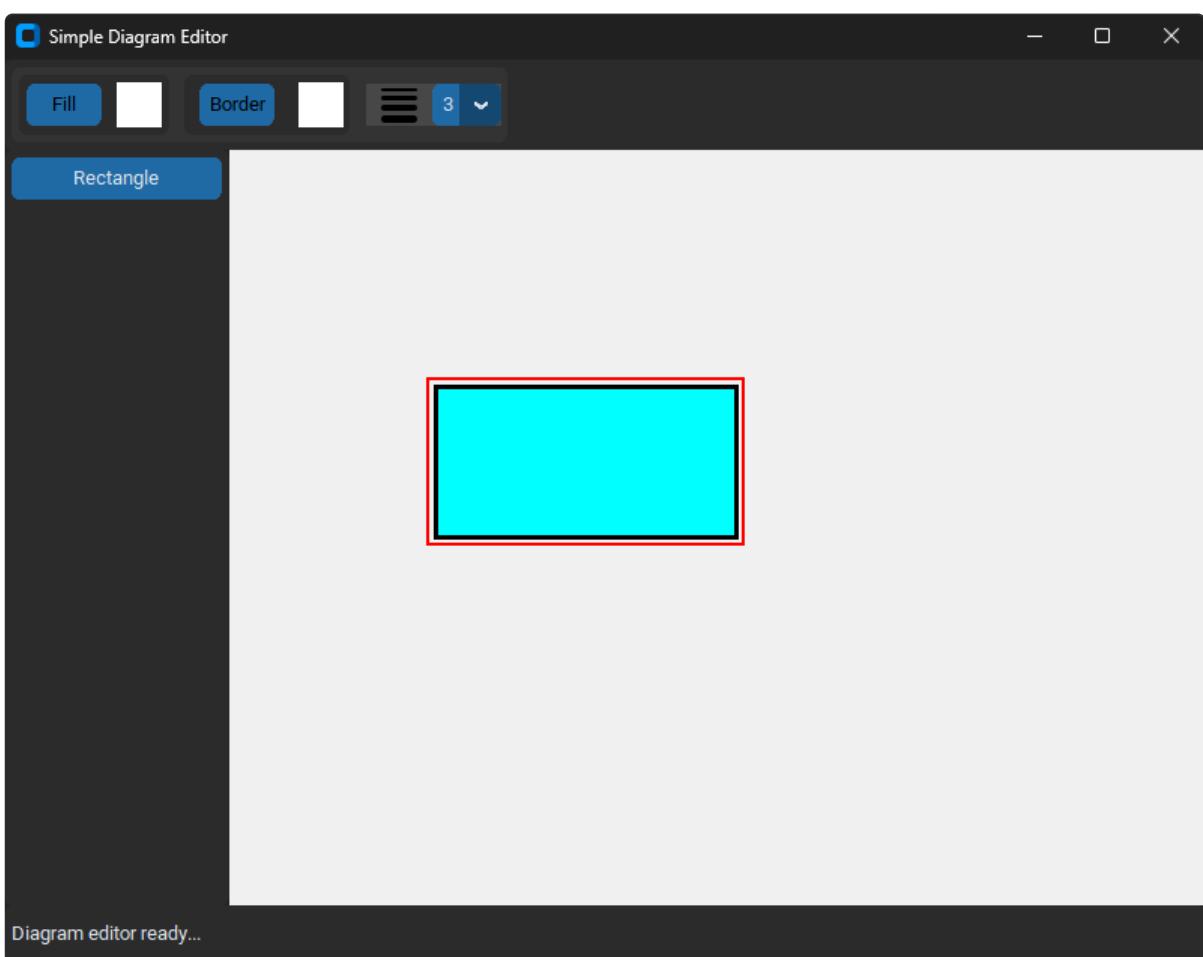
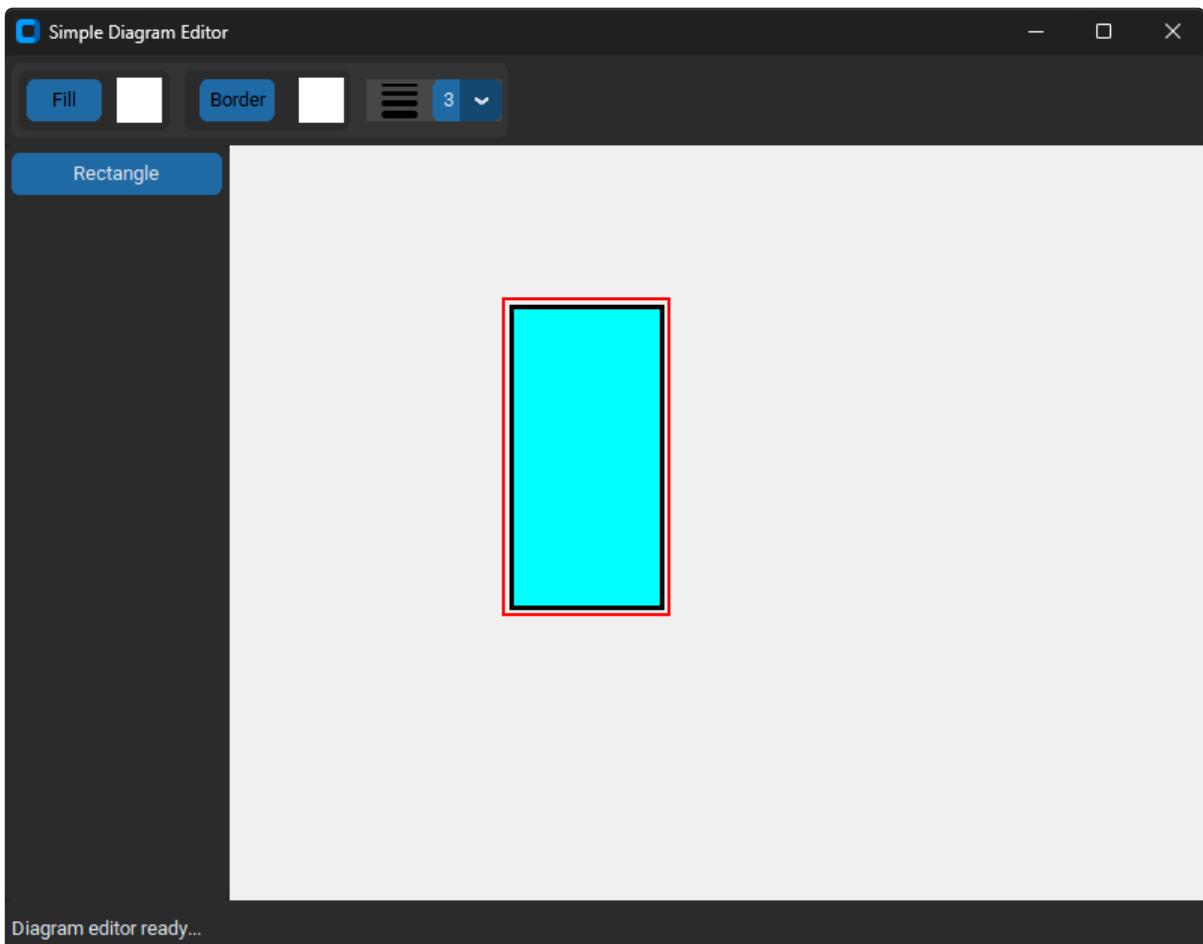
```

The keyboard needs to be instantiated in the Main Application File

simple_diagram_editor.py modifications

```
    . . .  
  
    self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)  
  
    self.keyboard = Keyboard(self, self.canvas) # Added new  
  
  
if __name__ == "__main__":  
    app = SimpleDiagramEditorApp()  
    app.mainloop()
```

In the rectangle class, I modified the size to 100x200 px. Run the program, create a rectangle, select the rectangle, press the 'r' key and verify that the rectangle rotates to 200x100.



This method works well for symmetric shapes. Non-symmetric shapes such as a triangle will need more work to define the shape at each rotation position. Text and images have a simple rotation feature. Lines will not be rotated because they can be resized to any rotation.

Draw Shape with Mouse

Currently, the program creates a rectangle with a fixed size and location on the canvas. In this section, we will modify the program to allow the user to select a shape from the Left Frame Menu and "draw" the shape with the mouse at any desired location and by dragging the left mouse button to any desired size. The Mouse Class has bindings for moving shapes with the mouse. We will add new bindings and event handlers to draw the shape and modify the button handler methods in the Left Frame Class to use the new draw bindings.

In the Mouse Class initializer, move the three mouse bindings to a method called `move_bind_mouse_events()`.

```
from Helper_Lib.point import Point

class Mouse:
    def __init__(self, a_canvas):
        """Class to manage mouse events"""
        self.canvas = a_canvas
        self.selected_shape = None

        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

    def move_bind_mouse_events(self):
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    . . .
```

Create a new method to unbind all mouse events and add a call to `unbind_mouse_events()` in the `move_bind_mouse_events()` method.

```

def unbind_mouse_events(self):
    self.canvas.unbind("<Button-1>")
    self.canvas.unbind("<B1-Motion>")
    self.canvas.unbind("<ButtonRelease-1>")

def move_bind_mouse_events(self):
    self.unbind_mouse_events()
    self.canvas.bind("<Button-1>", self.move_left_down)
    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

```

Next, add new mouse bindings for the draw event handlers.

```

def draw_bind_mouse_events(self):
    self.canvas.bind("<Button-1>", self.draw_left_down)
    self.canvas.bind("<B1-Motion>", self.draw_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

```

Implement the three draw event handlers: `draw_left_down()`, `draw_left_drag()`, and `draw_left_up()` to draw a rectangle when called. In the draw left down event handler, we simply capture the start x, y coordinate which is the location of the mouse click on the canvas and create a rectangle at the start x, y coordinates. In draw left drag, we assume that the upper-left corner (x_1, y_1) of the rectangle is set to the start x, y coordinate then we set the lower-right corner (x_2, y_2) coordinates to the mouse drag position and update the shape. This gives the effect that we are 'drawing' the shape with the mouse. The draw left down method, initializes the current shape variables to None, unbinds the mouse events, and sets the mouse bindings to move mouse events.

`mouse.py`

```

. . .

def move_left_up(self, _event):
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0

def draw_left_down(self, event):
    self.unselect_all()
    self.start.x = event.x
    self.start.y = event.y

```

```

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.canvas.shape_list.append(self.current_shape_obj)

def draw_left_drag(self, event):
    if self.current_shape_obj:
        shape = self.current_shape_obj
        x, y = event.x, event.y
        shape.x1, shape.y1 = self.start.x, self.start.y
        shape.x2, shape.y2 = x, y
        self.canvas.redraw_shapes()

def draw_left_up(self, _event):
    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.move_bind_mouse_events()

    . . .

```

In the Rectangle class, add a statement in the class initializer to create itself when the Rectangle object is created.

```

from Helper_Lib import Point

class Rectangle:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None
        self.sel_id = None
        self.is_selected = False

        self.create() # Added

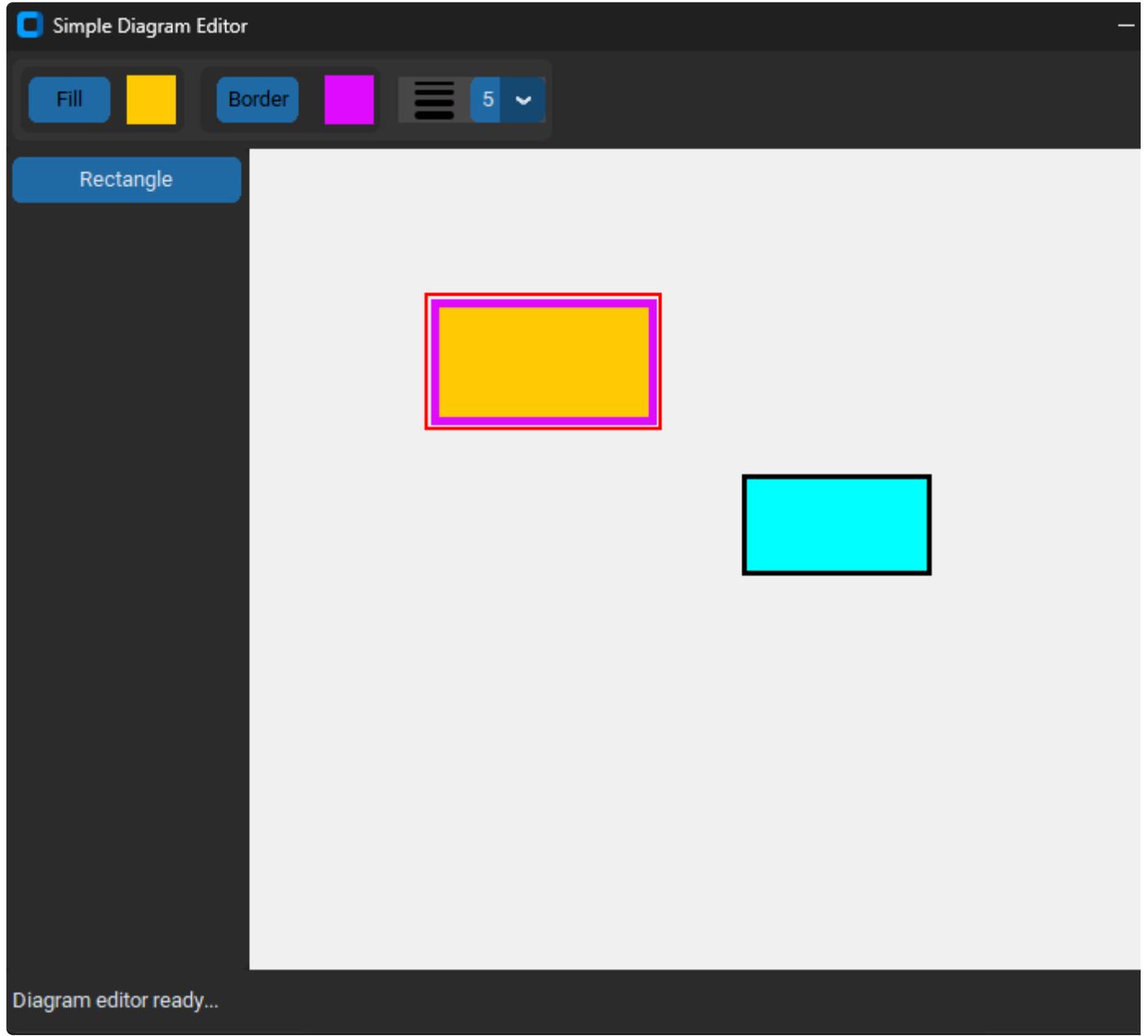
```

```
def create(self):
    """Create the shape once!"""
    self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

    self.sel_id = self.canvas.create_rectangle(self.x1-5, self.y1-5,
self.x2+5, self.y2+5, fill=None,
                                         outline="red", width=2)
    self.canvas.itemconfig(self.sel_id, state='hidden')
```

Run the program, select the Rectangle button from the Shape Menu in the Left Frame. Nothing happens. Click on the canvas and drag the mouse to create a rectangle. Draw a second rectangle and verify that you can move it, change the fill color, border color, and

border width.

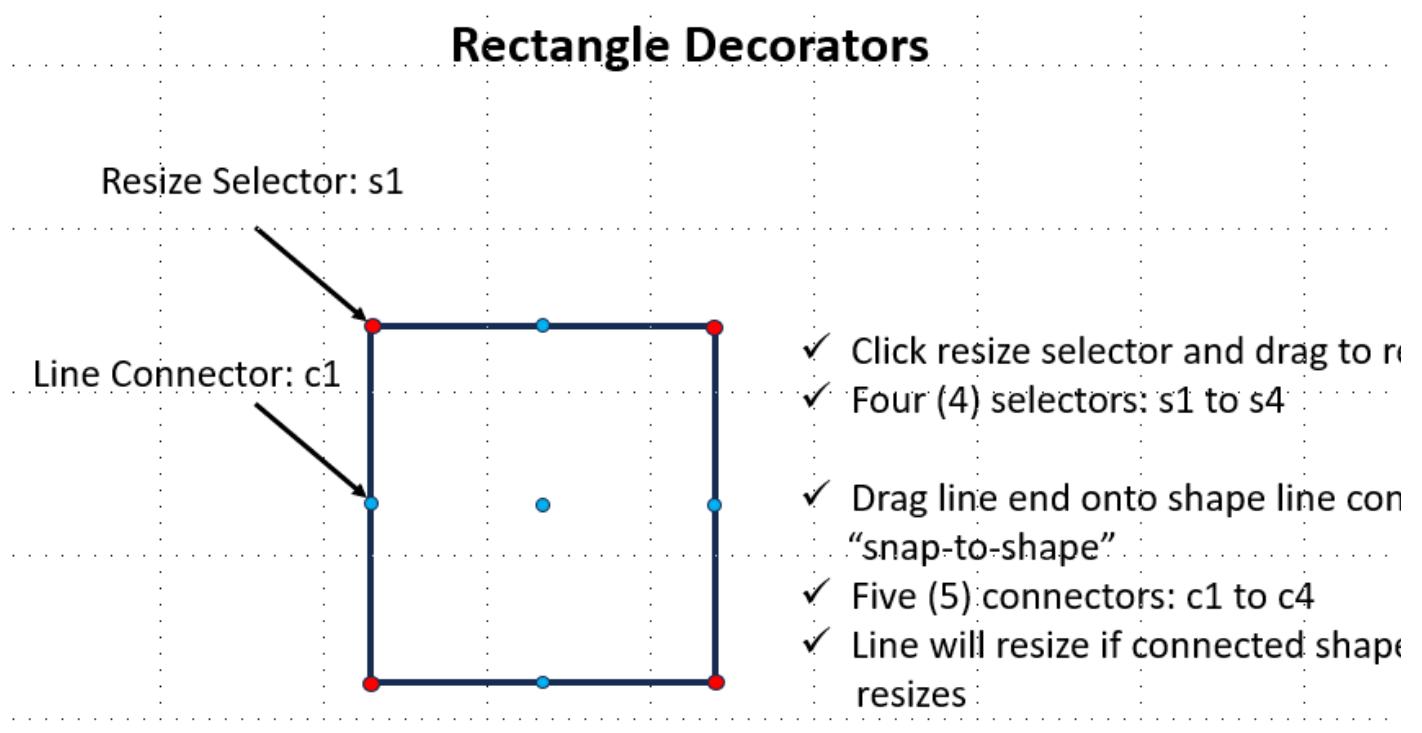


We can now draw shapes with the mouse.

Resize Shape with Mouse

We want to add a program "feature" that allows the user to resize an existing shape with the mouse. The user selects a corner of the shape and "drags" it with the left mouse button to resize the shape. We will add new mouse bindings and event handlers to resize the shape. How do we know that the user wants to resize the shape? We will modify the shape decorators to add selectors which are red ovals and if the user clicks on one of the selectors, we bind the mouse to the resize event handlers.

The following figure shows the two type of rectangle decorators we will need: "selectors" for resize mouse events and "connectors" for line connections to the shape (future capability).



We will create classes for each of the decorator types.

Selector Class

Create a new file called `selector.py` in the `Shape_Lib`. Create a new class called `Selector` and initialize it with the canvas, a name, and x, y coordinates. Add an attribute called `radius` and set it to 5 px.

```
class Selector:  
    def __init__(self, canvas, name, x, y):  
        self.canvas = canvas  
        self.name = name  
        self.x = x  
        self.y = y  
  
        self.radius = 5  
        self.id = None
```

Add a `create()` method that creates a red oval with outline color of black and width of 2 px. Call `create()` from the initializer.

```

class Selector:
    def __init__(self, a_canvas, name, x, y):
        self.canvas = a_canvas
        self.name = name
        self.x = x
        self.y = y

        self.radius = 5
        self.id = None
        self.create()

    def create(self):
        """Create the shape here"""
        sel_points = [self.x - self.radius, self.y - self.radius, self.x +
        self.radius, self.y + self.radius]
        self.canvas.create_oval(sel_points, fill="red", outline="black",
width=2)

```

Add an update() method to update the position of the selector.

```

def update(self):
    """Update the shape here"""
    self.canvas.coords(self.x - self.radius, self.y - self.radius, self.x +
    self.radius, self.y + self.radius)

```

Add a selector hit test method that will be used to determine if the user clicks on it with the mouse.

```

def selector_hit_test(self, event_x, event_y):
    x1, y1 = self.x - self.radius, self.y - self.radius
    x2, y2 = self.x + self.radius, self.y + self.radius
    if x1 <= event_x <= x2 and y1 <= event_y <= y2:
        return True
    else:
        return False

```

Add a `__repr__()` to override the `print()` method with a custom print string.

```

def __repr__(self):
    return "Selector: " + self.name + " (" + str(self.x) + ", " +

```

```
str(self.y) + ")"
```

Add a main test section to test the selector locally before integrating it into the main program.

```
if __name__ == "__main__":
    import customtkinter as ctk

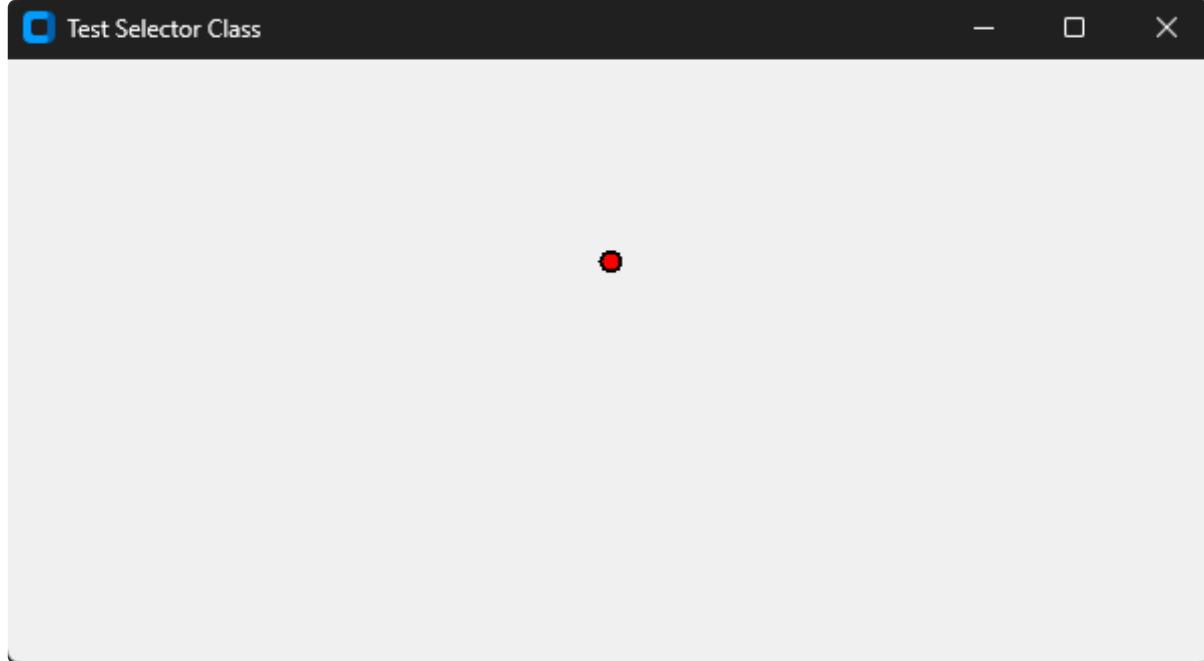
    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Selector Class")

    canvas = ctk.CTkCanvas(app)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    sel = Selector(canvas, "test", 300, 100)
    print(sel)
    print("Hit test: ", sel.selector_hit_test(300, 100))

    app.mainloop()
```

Run the Selector Class program



Console Output

```
Selector: test (300, 100)
```

```
Hit test: True
```

Here is the complete selector class without the test code.

```
class Selector:  
    def __init__(self, a_canvas, name, x, y):  
        self.canvas = a_canvas  
        self.name = name  
        self.x = x  
        self.y = y  
  
        self.radius = 5  
        self.id = None  
        self.create()  
  
    def create(self):  
        """Create the shape here"""  
        sel_points = [self.x - self.radius, self.y - self.radius, self.x +  
self.radius, self.y + self.radius]  
        self.canvas.create_oval(sel_points, fill="red", outline="black",  
width=2)  
  
    def update(self):  
        """Update the shape here"""  
        self.canvas.coords(self.x - self.radius, self.y - self.radius, self.x +  
self.radius, self.y + self.radius)  
  
    def selector_hit_test(self, event_x, event_y):  
        x1, y1 = self.x - self.radius, self.y - self.radius  
        x2, y2 = self.x + self.radius, self.y + self.radius  
        if x1 <= event_x <= x2 and y1 <= event_y <= y2:  
            return True  
        else:  
            return False  
  
    def __repr__(self):  
        return "Selector: " + self.name + " (" + str(self.x) + ", " +  
str(self.y) + ")"
```

Seems like a lot of work for one small red oval. We will see the power of this class as we add it to the Rectangle Class to define its selectors. We need to replace the simple red rectangle selector, with four selectors defined by our new Selector Class and position them at the four corners of the rectangle shape. First, we need to create the selectors, hide them until the rectangle is selected, and update their position if the rectangle is moved, rotated, or resized.

In the Rectangle Class, import the Selector Class
rectangle.py

```
from Helper_Lib import Point
from Shape_Lib.selector import Selector # Added
```

In the Rectangle Class initializer, create a selector list called `self.sel_list`. Also, create four select id variables and call a create_selectors method that we will code soon.

```
def __init__(self, a_canvas, x1, y1, x2, y2):
    self.canvas = a_canvas
    self.x1 = x1
    self.y1 = y1
    self.x2 = x2
    self.y2 = y2
    self.fill_color = "cyan"
    self.border_color = "black"
    self.border_width = 3

    self.id = None
    self.is_selected = False
    self.sel_list = [] # Added

    self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None # Added

    self.create_shape()
    self.create_selectors() # Added
```

Modify the update() method to toggle the visibility of the selectors based on the `is_selected` variable.

```

def update(self):
    """Update the shape here"""
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, outline=self.border_color)
    self.canvas.itemconfig(self.id, width=self.border_width)

    self.update_selectors()
    if self.is_selected:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

```

Add the `create_selectors()` method to create four selectors, add them to the `sel_list`, and set visibility to off ('hidden').

```

def create_selectors(self):
    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
    self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
    self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
    self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

```

Add a new method called `update_selectors()` which will update the position of all selectors based on the position of the parent rectangle.

```

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y1
    self.s2_id.update()

    self.s3_id.x, self.s3_id.y = self.x2, self.y2

```

```
    self.s3_id.update()

    self.s4_id.x, self.s4_id.y = self.x1, self.y2
    self.s4_id.update()
```

Add a new method called `resize()` that changes the rectangle coordinates based on the selector name and the mouse offset position.

```
def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        self.x1, self.y2 = x1, y2
```

Add a new method called `check_selector_hit()` that will check to see if a selector has been selected by the user.

```
def check_selector_hit(self, x, y):
    for sel in self.sel_list:
        if sel.selector_hit_test(x, y):
            return sel
    return None
```

No changes to the `rotate()` or `__repr__()` methods.

Here is the complete Rectangle Class.

```
from Helper_Lib import Point
from Shape_Lib.selector import Selector


class Rectangle:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None
        self.is_selected = False
        self.sel_list = []
        self.selector = None

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None

    self.create_shape()
    self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                            fill=self.fill_color,
                                            outline=self.border_color,
                                            width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
```

```

        self.canvas.itemconfig(s.id, state='hidden')

def create_selectors(self):
    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
    self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
    self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
    self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y1
    self.s2_id.update()

    self.s3_id.x, self.s3_id.y = self.x2, self.y2
    self.s3_id.update()

    self.s4_id.x, self.s4_id.y = self.x1, self.y2
    self.s4_id.update()

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2

```

```

        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        self.x1, self.y2 = x1, y2

def check_selector_hit(self, x, y):
    for sel in self.sel_list:
        if sel.selector_hit_test(x, y):
            return sel
    return None

def __repr__(self):
    return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

if __name__ == "__main__":
    import customtkinter as ctk
    from UI_Lib import Canvas

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Rectangle Class")

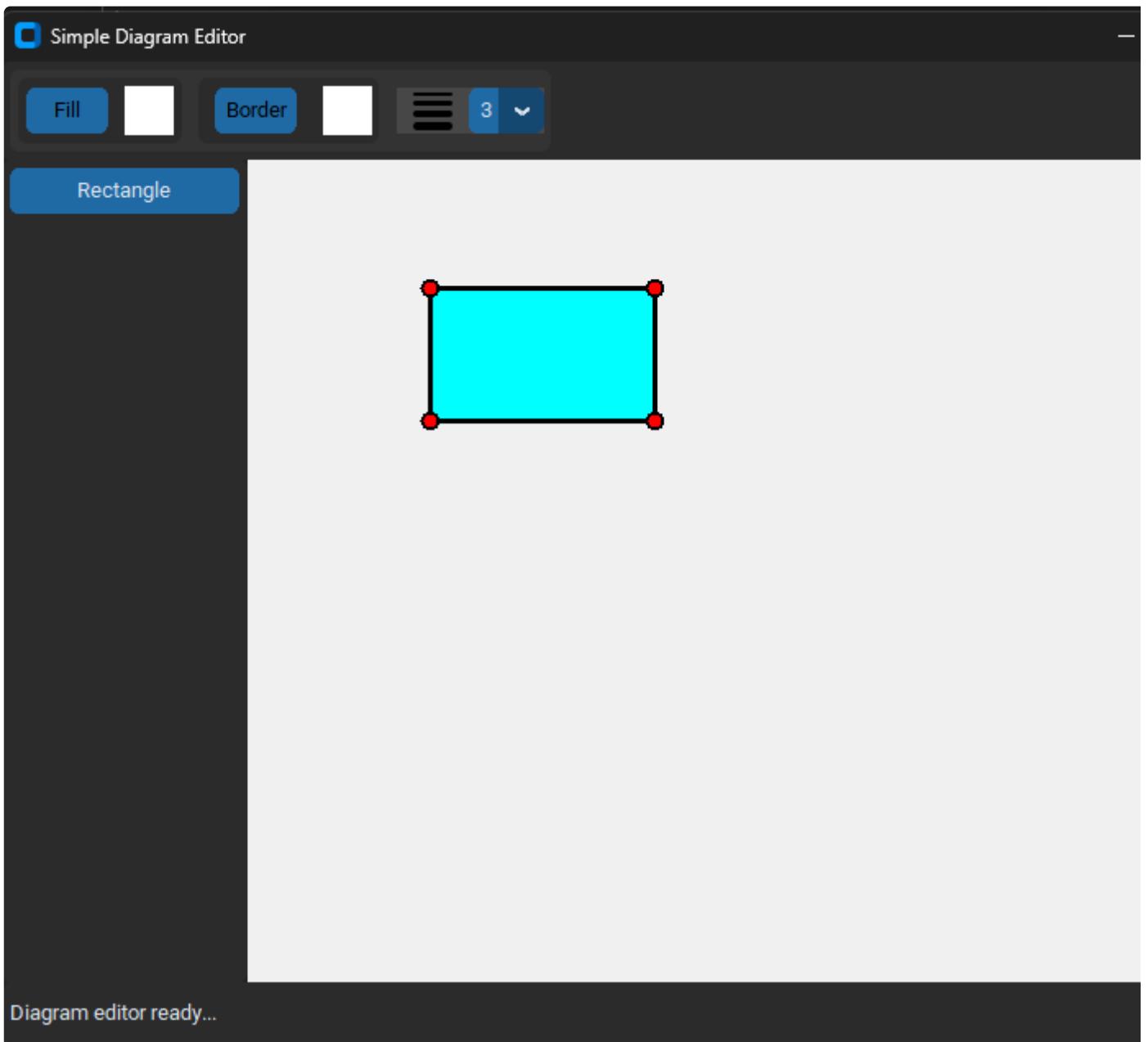
    canvas = Canvas(app)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    canvas.mouse.current_shape = "rectangle"
    canvas.mouse.draw_bind_mouse_events()

    app.mainloop()

```

Run the main program, draw a rectangle, select the rectangle, and confirm that the selectors can be toggled on and off.



Next we need to modify the Mouse Class so that it can detect a "hit" on a selector and change to resize mouse bindings which allow the user to resize the rectangle by dragging the selector to a new position. This needs to work for any of the four selectors.

Modify the Mouse Class to add resize bindings and resize event handlers

```
...  
  
def draw_bind_mouse_events(self):  
    self.canvas.bind("<Button-1>", self.draw_left_down)  
    self.canvas.bind("<B1-Motion>", self.draw_left_drag)  
    self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)
```

```
def bind_resize_mouse_events(self): # Added new mouse bindings
    self.canvas.bind("<Button-1>", self.resize_left_down)
    self.canvas.bind("<B1-Motion>", self.resize_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

. . .
```

Add the resize event handlers to the Mouse Class. These handlers are very similar to the draw event handlers. The resize left down method captures the shape coordinates and calculates all x and y offsets. The resize left drag method creates an offsets list and calls the shape `resize()` method that we added to the Rectangle Class earlier and calls the canvas `redraw_shapes()` method. The resize left up method resets the offsets to 0 and changes the mouse bindings for move operations.

```
. . .
```

```
def draw_left_up(self, _event):
    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.move_bind_mouse_events()

def resize_left_down(self, event): # New resize event handler
    if self.selected_shape:
        shape = self.selected_shape
        x1, y1 = shape.x1, shape.y1
        x2, y2 = shape.x2, shape.y2
        self.offset1.x = event.x - x1
        self.offset1.y = event.y - y1
        self.offset2.x = event.x - x2
        self.offset2.y = event.y - y2

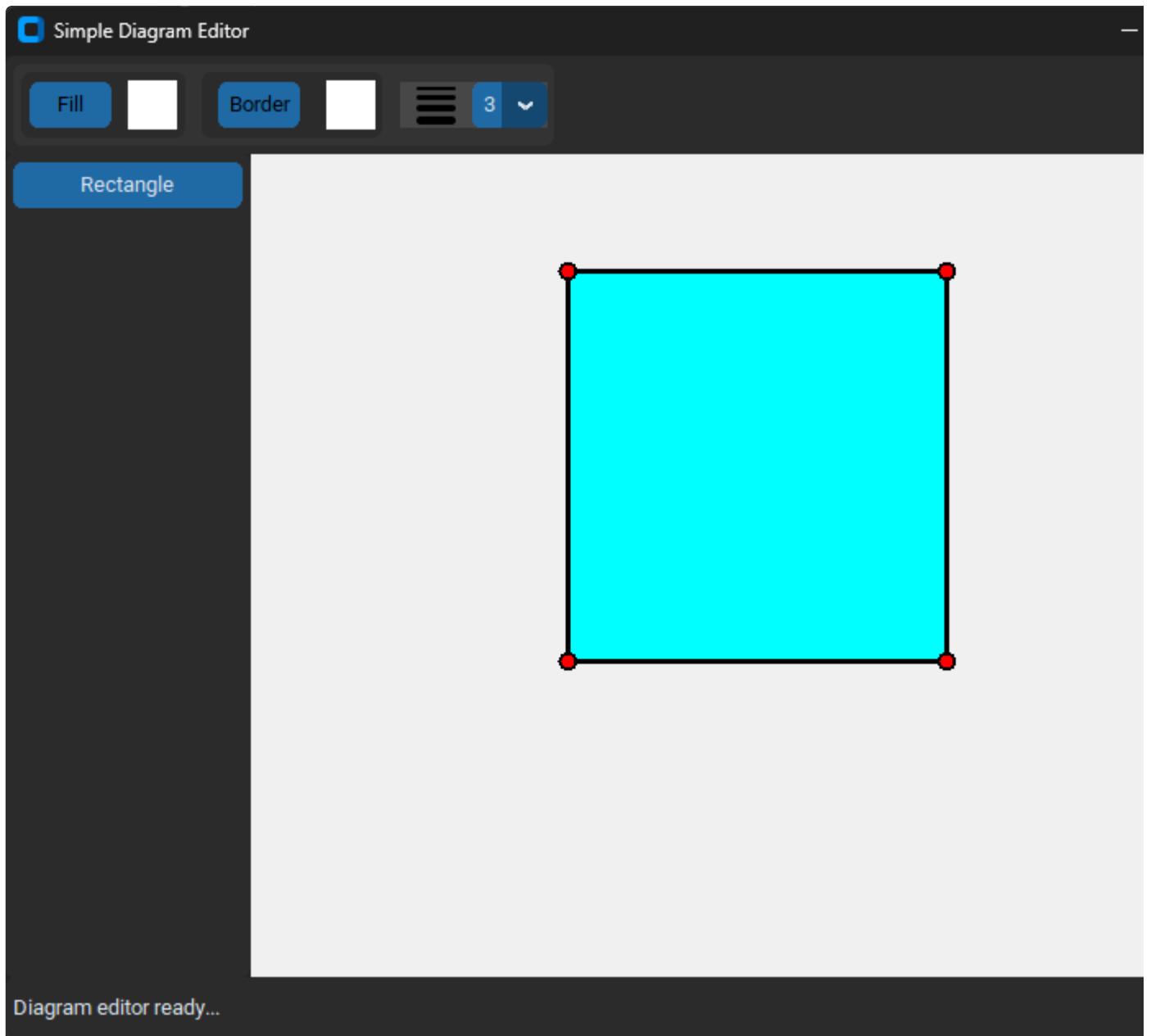
def resize_left_drag(self, event): # New resize event handler
    if self.selected_shape:
        offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
self.offset2.y]
        self.selected_shape.resize(offsets, event)
        self.canvas.redraw_shapes()

def resize_left_up(self, _event): # New resize event handler
    self.offset1.x, self.offset1.y = 0, 0
    self.offset2.x, self.offset2.y = 0, 0
    self.canvas.mouse.unbind_mouse_events()
```

```
self.canvas.mouse.move_bind_mouse_events()
```

```
... . . .
```

Run the main program, draw a rectangle, select the rectangle to show the selectors, drag one or more of the selectors to confirm that you can resize the rectangle.



Refactor Test Code

Lets remove the test code section from:

- Rectangle Class
- Mouse Class
- Canvas Class
- Selector Class
- Point Class

Why remove the test code? The classes are fairly mature at this point and removing the test code means that we don't have to maintain it as the development progresses. We will add the test code section to any class that would benefit from local testing before integration into the main application.

After deleting the test code, run the program and confirm that it runs without errors.

Straight Line Class

We will create three line classes for the Diagram Editor App:

- Straight Line Class
- Segment Line Class - three segments
- Elbow Line Class - two segments

Lines are drawn between two endpoints: x1, y1 and x2, y2. Lines can be drawn on the canvas using `canvas.create_line()` method in a Line Class. In the Shape_Lib directory create a new file called `straight_line.py`.

In the new file, import the Selector Class

```
from Shape_Lib.selector import Selector
```

Name the class: `straightLine` and initialize it with the canvas and the x1, y1, x2, y2 coordinates. Add the selector, id, is_selected and `sel_list` attributes. Set the fill color to black and width to 3. Recall that lines do not have an outline color. Create two selector id variables called `s1_id` and `s2_id`. Call the create shape and create selectors methods that we will add shortly.

```

class StraightLine:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.selector = None
        self.id = None
        self.is_selected = False
        self.sel_list = []

        self.fill_color = "black"
        self.border_width = 3

        self.s1_id, self.s2_id = None, None

    self.create_shape()
    self.create_selectors()

```

Add the `create_shape()` method that creates a Line Shape and stores a reference to the shape id.

```

def create_shape(self):
    """Create the shape here"""
    self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
                                      fill=self.fill_color,
                                      width=self.border_width)

```

Add the `update()` method that updates the location, fill color, and border width of the line. It also call the `update_selectors()` method and sets the visibility of the selectors based on the `is_selected` variable.

```

def update(self):
    """Update the shape here"""
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, width=self.border_width)

    self.update_selectors()

```

```

if self.is_selected:
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='normal')
else:
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

```

Add the `create_selectors()` method that creates two selectors:

- Name = 'begin' at the x1, y1 point
- Name = 'end' at the x2, y2 point

It then adds the selector id's to the selector list and sets the visibility to hidden. Note that the line selectors will be shown when the line is selected.

```

def create_selectors(self):
    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
    self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

    self.sel_list = [self.s1_id, self.s2_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

```

Add the `update_selectors()` method that updates the position of the two selectors based on the line position coordinates.

```

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y2
    self.s2_id.update()

```

Add a `resize()` method that move the end of the line associated with the selector name, either 'begin' or 'end'.

```

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets

```

```

if self.selector == "end":
    x2 = event.x - offset_x2
    y2 = event.y - offset_y2
    self.x2, self.y2 = x2, y2
elif self.selector == "begin":
    x1 = event.x - offset_x1
    y1 = event.y - offset_y1
    self.x1, self.y1 = x1, y1

```

Add a new method called `check_selector_hit()` that iterates over the selector list and calls the selector hit test. If hit, the method returns the selector otherwise it returns None.

```

def check_selector_hit(self, x, y):
    for sel in self.sel_list:
        if sel.selector_hit_test(x, y):
            return sel
    return None

```

Add a `__repr__()` method to provide a custom string when the line is printed.

```

def __repr__(self):
    return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

Here is the complete Straight Line Class.

```

from Shape_Lib.selector import Selector

class StraightLine:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.selector = None
        self.id = None
        self.is_selected = False
        self.sel_list = []

```

```

        self.fill_color = "black"
        self.border_width = 3

        self.s1_id, self.s2_id = None, None

        self.create_shape()
        self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
                                         fill=self.fill_color,
                                         width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

```

```

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1

def check_selector_hit(self, x, y):
    for sel in self.sel_list:
        if sel.selector_hit_test(x, y):
            return sel
    return None

def __repr__(self):
    return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

Modify the Left Frame Class to add a new button called Line and a button handler to set the current shape variable to "straight".

`left_frame.py`

```

import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        # self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self, text="Rectangle",
command=self.create_rect)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

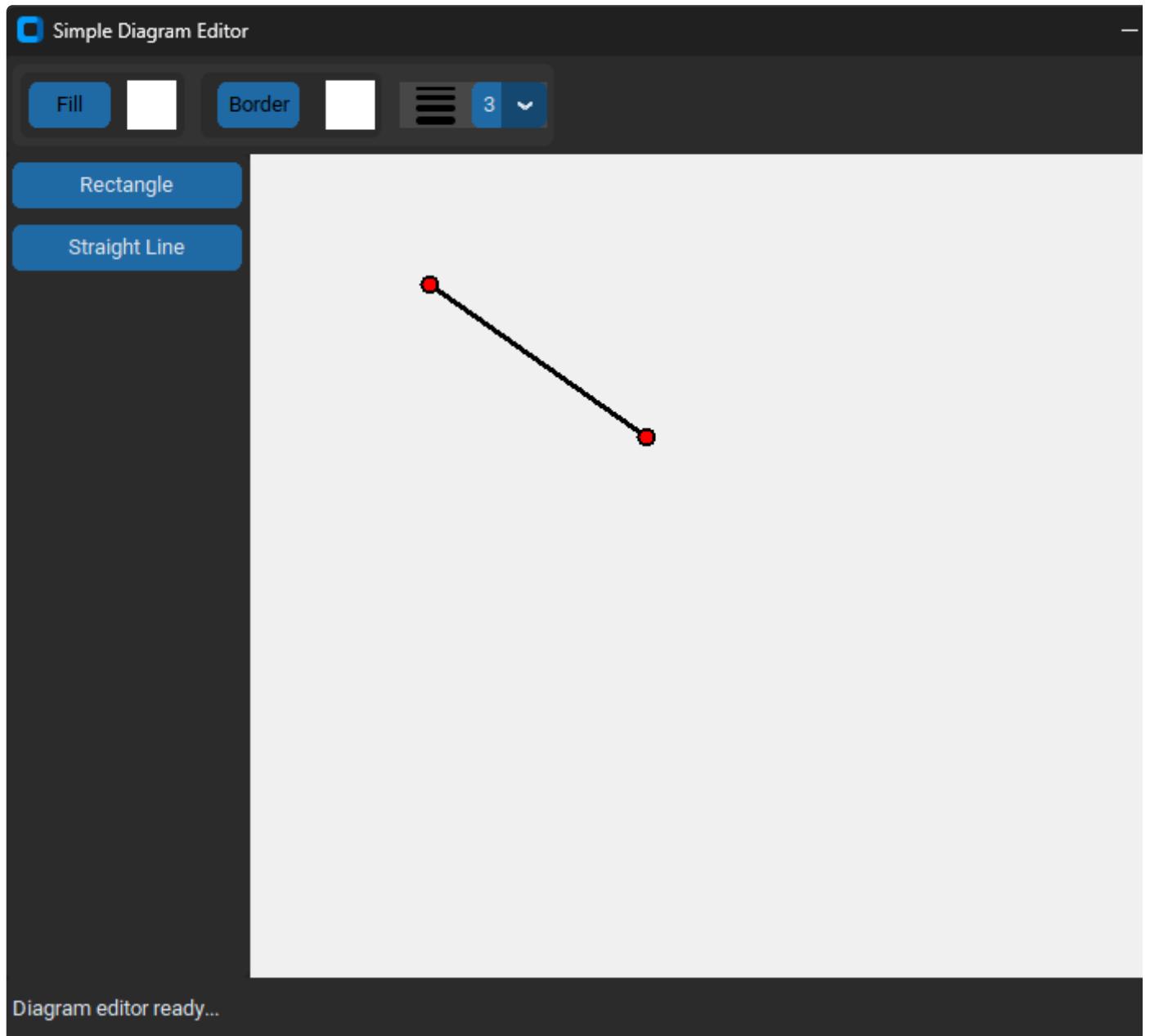
        straight_line_button = ctk.CTkButton(self, text="Straight Line",
command=self.create_straight_line) # Added
        straight_line_button.pack(side=ctk.TOP, padx=5, pady=5) # Added

```

```
# Left frame button handlers
def create_rect(self):
    self.canvas.mouse.current_shape = "rectangle"
    self.canvas.mouse.draw_bind_mouse_events()

def create_straight_line(self): # Added new method
    self.canvas.mouse.current_shape = "straight"
    self.canvas.mouse.draw_bind_mouse_events()
```

Run the main program, draw a line, and verify that you can move and resize the line with the mouse.



Connector Class

Shape Connectors

We will now implement the second shape decorator which is a connector represented graphically by a cyan oval. What is a connector used for? We want to add the program feature to draw lines between connectors on shapes such that if the shape is move, rotated, or resized the "connected" line will automatically resize.

Here is a summary of the changes needed to implement the connector feature:

- ✓ Create a new Connector Class
- ✓ Create a new Connection Class
- ✓ Modify the Rectangle Class to create and update Connectors
- ✓ Add a new `line_list` to the Rectangle Class to store "connections" to it
- ✓ Modify the mouse class for line drawing to check to see if a line is drawn from or to a connector on a shape. If there is a connector hit, add a "connection" to the parent shape's line list.
- ✓ Modify the Rectangle Class to move any connected lines if a shape is moved

Connector Class

Create a new file called `connector.py` in the Shape Library. The connector class is very similar to the selector class so I will show the code without explanation. If you need further explanation, look at the explanation in the [Resize Shape with Mouse](#) section.

```
class Connector:  
    def __init__(self, a_canvas, name, _x, _y):  
        self.canvas = a_canvas  
        self.name = name  
        self.x = _x  
        self.y = _y  
  
        self.radius = 5  
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -  
                                             self.radius,  
                                             self.x + self.radius, self.y +  
                                             self.radius)  
  
        self.id = None  
        self.create()
```

```

def create(self):
    """Create the shape here"""
    sel_points = [self.x1, self.y1, self.x2, self.y2]
    self.id = self.canvas.create_oval(sel_points, fill="cyan",
outline="black", width=2, tags='selector')

def update(self):
    """Update the shape here"""
    self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
self.radius,
                                         self.x + self.radius, self.y +
self.radius)
    sel_points = [self.x1, self.y1, self.x2, self.y2]
    self.canvas.coords(self.id, sel_points)

def connector_hit_test(self, event_x, event_y):
    if self.x1 <= event_x <= self.x2 and self.y1 <= event_y <= self.y2:
        return True
    else:
        return False

def __repr__(self):
    return ("Connector: " + self.name + " (" + str(self.x1) + ", " +
str(self.y1) + ")" +
        " (" + str(self.x2) + ", " + str(self.y2) + ")")

```

Add a new file called connection.py to the Shape Library. We will define a connection as the connector, line, and a line end.

```

class Connection:
    def __init__(self, conn_obj, line_obj, line_end):
        self.connector_obj = conn_obj
        self.line_obj = line_obj
        self.line_end = line_end      # "begin" or "end"

    def __repr__(self):
        return "Connection Object: " + self.connector_obj.name + \
               " Connection Object Location: " + str(self.connector_obj.x) + ", " +
str(self.connector_obj.y) + \
               " Line Object Points: " + str(self.line_obj.points) + \
               " Line End: " + self.line_end

```

Modify the Rectangle Class to add five connections to the center of the rectangle and the centers of each side.

```
from Helper_Lib import Point
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector # Added

class Rectangle:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None
        self.is_selected = False
        self.is_drawing = False # Added
        self.sel_list = []
        self.conn_list = [] # Added
        self.line_list = [] # Added
        self.selector = None

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
        None, None, None, None # Added

        self.create_shape()
        self.create_selectors()
        self.create_connectors() # Added

    def create_shape():
        """Create the shape here"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
                                             self.y2,
                                             fill=self.fill_color,
                                             outline=self.border_color,
                                             width=self.border_width)

    def update(self):
        """Update the shape here"""


```

```

        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

    self.update_selectors()
    if self.is_selected:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    self.update_connectors() # Added
    if self.is_drawing: # Added
        for c in self.conn_list: # Added
            self.canvas.itemconfig(c.id, state='normal') # Added
    else: # Added
        for c in self.conn_list: # Added
            self.canvas.itemconfig(c.id, state='hidden') # Added

    self.move_connected_lines() # Added

def create_selectors(self):
    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
    self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
    self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
    self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y1
    self.s2_id.update()

    self.s3_id.x, self.s3_id.y = self.x2, self.y2
    self.s3_id.update()

    self.s4_id.x, self.s4_id.y = self.x1, self.y2

```

```

    self.s4_id.update()

def create_connectors(self): # Added new method
    """Create connectors here"""
    # Calculate the shape geometry
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
    self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
    self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
    self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
    self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

    self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
    for c in self.conn_list:
        self.canvas.itemconfig(c.id, state='hidden')

def update_connectors(self): # Added new method
    """Update the position of all connectors here"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id.x, self.c1_id.y = center.x, center.y
    self.c1_id.update()

    self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
    self.c2_id.update()

    self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
    self.c3_id.update()

    self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
    self.c4_id.update()

    self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
    self.c5_id.update()

```

```

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        self.x1, self.y2 = x1, y2

def check_selector_hit(self, x, y):
    for sel in self.sel_list:
        if sel.selector_hit_test(x, y):
            return sel
    return None

def check_connector_hit(self, x, y): # Added new method
    for conn in self.conn_list:
        if conn.connector_hit_test(x, y):
            return conn
    return None

def move_connected_lines(self): # Added new method
    for connection in self.line_list:
        for connector in self.conn_list:
            if connector == connection.connector_obj:
                # print(connector, connection.line_obj, "Match")
                if connection.line_end == "begin":
                    connection.line_obj.x1 = connector.x
                    connection.line_obj.y1 = connector.y
                elif connection.line_end == "end":

```

```

        connection.line_obj.x2 = connector.x
        connection.line_obj.y2 = connector.y

    def __repr__(self):
        return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

```

Modify the Left Frame Class to show connectors if the user draws a straight line.

```

import customtkinter as ctk

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        # self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self, text="Rectangle",
command=self.create_rect)
        rect_button.pack(side=ctk.TOP, padx=5, pady=5)

        straight_line_button = ctk.CTkButton(self, text="Straight Line",
command=self.create_straight_line)
        straight_line_button.pack(side=ctk.TOP, padx=5, pady=5)

    # Left frame button handlers
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_straight_line(self):
        self.canvas.mouse.current_shape = "straight"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def show_connectors(self):
        for s in self.canvas.shape_list:

```

```

        s.is_drawing = True
        self.canvas.redraw_shapes()

    def hide_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = False
        self.canvas.redraw_shapes()

```

Modify the Mouse Class to check for connector hits when drawing a line.

```

. . .

def draw_left_down(self, event):
    self.unselect_all()
    self.start.x = event.x
    self.start.y = event.y

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "straight":
        self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y) # Added

    if self.current_shape_obj is not None: # Changed
        self.canvas.shape_list.append(self.current_shape_obj) # Changed

. . .

def draw_left_up(self, event):
    if isinstance(self.current_shape_obj, StraightLine): # Added
        self.select_connector(self.current_shape_obj, "end", event.x,
event.y) # Added
        self.current_shape = None
        self.current_shape_obj = None
        self.unbind_mouse_events()
        self.move_bind_mouse_events()

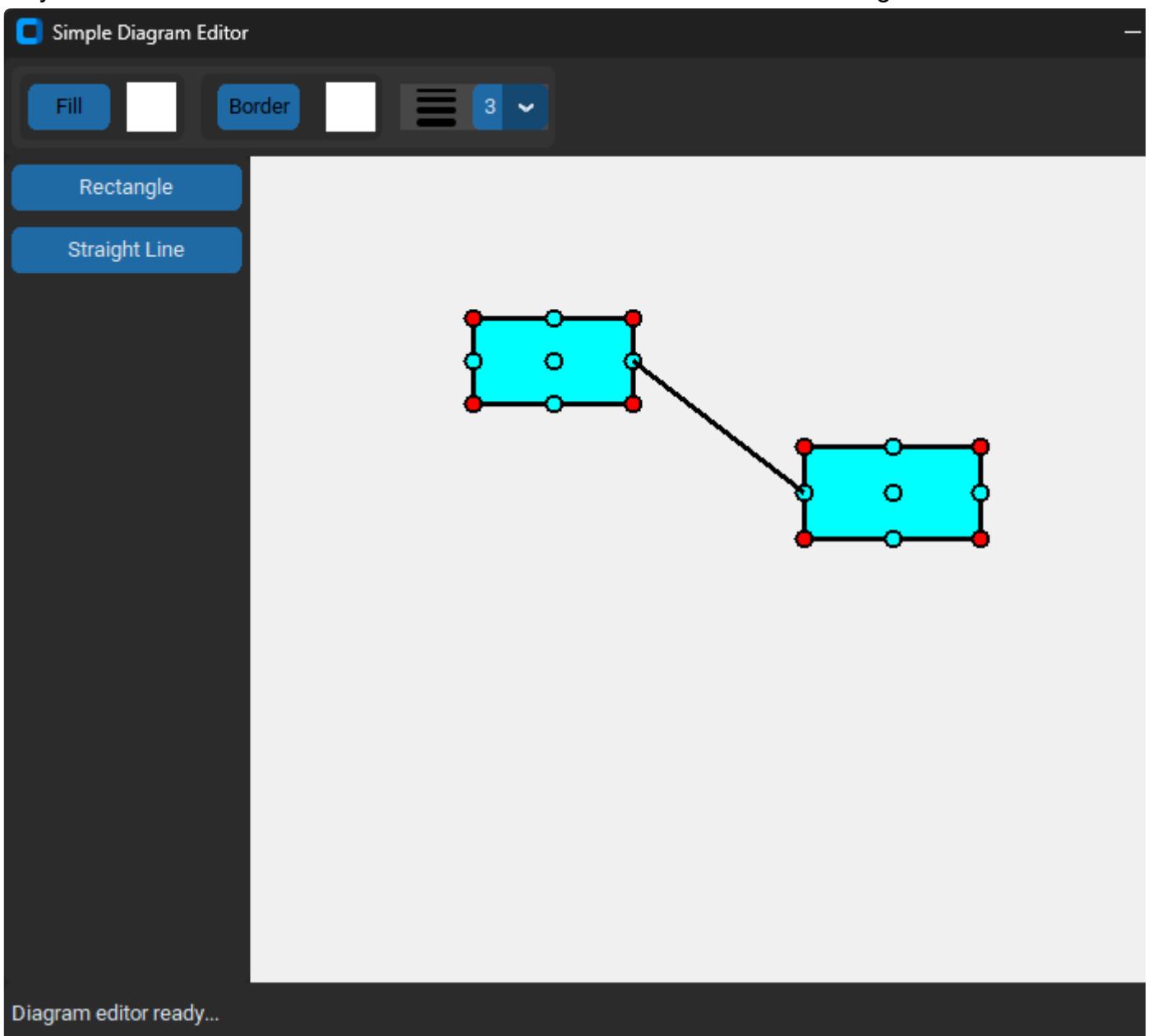
. . .

def select_connector(self, line_obj, line_end, x, y): # Added new method

```

```
for shape in self.canvas.shape_list:  
    conn = shape.check_connector_hit(x, y)  
    if conn:  
        if line_end == "begin":  
            line_obj.x1, line_obj.y1 = conn.x, conn.y  
        elif line_end == "end":  
            line_obj.x2, line_obj.y2 = conn.x, conn.y  
        a_conn = Connection(conn, line_obj, line_end)  
        shape.line_list.append(a_conn)  
        self.canvas.redraw_shapes()  
return
```

Run the program, draw two rectangles, draw one line between connectors on the two rectangle, select and move a rectangle, verify that the attached line automatically resizes to stay connected. Note that the connectors will be hidden when a new rectangle is drawn.



Shape Button Frames

In this section, we will create two custom frames for shape buttons and line button. The buttons will be images in a grid layout so it looks like a button array.

Shape Button Frame

The shape button frame will have five buttons for rectangle, oval, triangle, text, image. Create a new file called shape_button_frame.py in the UI_Lib.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image

class ShapeButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                           dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                           size=(24, 24))
        rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,
command=self.create_rect)
        rect_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(rect_button, msg="Rectangle")

        oval_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                           dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                           size=(24, 24))
        oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30)
        oval_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
```

```
ToolTip(oval_button, msg="Oval")

tri_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
size=(24, 24))
tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30)
tri_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
ToolTip(tri_button, msg="Triangle")

text_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/text.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
size=(24, 24))
text_button = ctk.CTkButton(self, text="", image=text_image, width=30)
text_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
ToolTip(text_button, msg="Text")

pic_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
size=(24, 24))
pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30)
pic_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
ToolTip(pic_button, msg="Image")

# Shape button handlers
def create_rect(self):
    self.canvas.mouse.current_shape = "rectangle"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def hide_connectors(self):
    for s in self.canvas.shape_list:
        s.is_drawing = False
    self.canvas.redraw_shapes()
```

Line Button Frame

The line button frame will have three buttons for straight line, segment line, and elbow line. Create a new file called line_button_frame.py in the UI_Lib.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image

class LineButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        straight_image = ctk.CTkImage(light_image=Image.open
                                       ("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                       dark_image=Image.open
                                       ("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                       size=(24, 24))
        straight_button = ctk.CTkButton(self, text="", image=straight_image,
                                        width=30,
                                        command=self.create_straight_line)
        straight_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(straight_button, msg="Straight Line")

        segment_image = ctk.CTkImage(light_image=Image.open
                                       ("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                       dark_image=Image.open
                                       ("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                       size=(24, 24))
        segment_button = ctk.CTkButton(self, text="", image=segment_image,
                                       width=30)
        segment_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(segment_button, msg="Segment Line")

        elbow_image = ctk.CTkImage(light_image=Image.open
```

```

("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
    dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
    size=(24, 24))
elbow_button = ctk.CTkButton(self, text="", image=elbow_image,
width=30)
elbow_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
ToolTip(elbow_button, msg="Elbow Line")

# Shape button handlers
def create_straight_line(self):
    self.canvas.mouse.current_shape = "straight"
    self.show_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def show_connectors(self):
    for s in self.canvas.shape_list:
        s.is_drawing = True
    self.canvas.redraw_shapes()

```

Modify the Left Frame Class to use the new custom button frames.

```

import customtkinter as ctk
from UI_Lib.shape_button_frame import ShapeButtonFrame
from UI_Lib.line_button_frame import LineButtonFrame

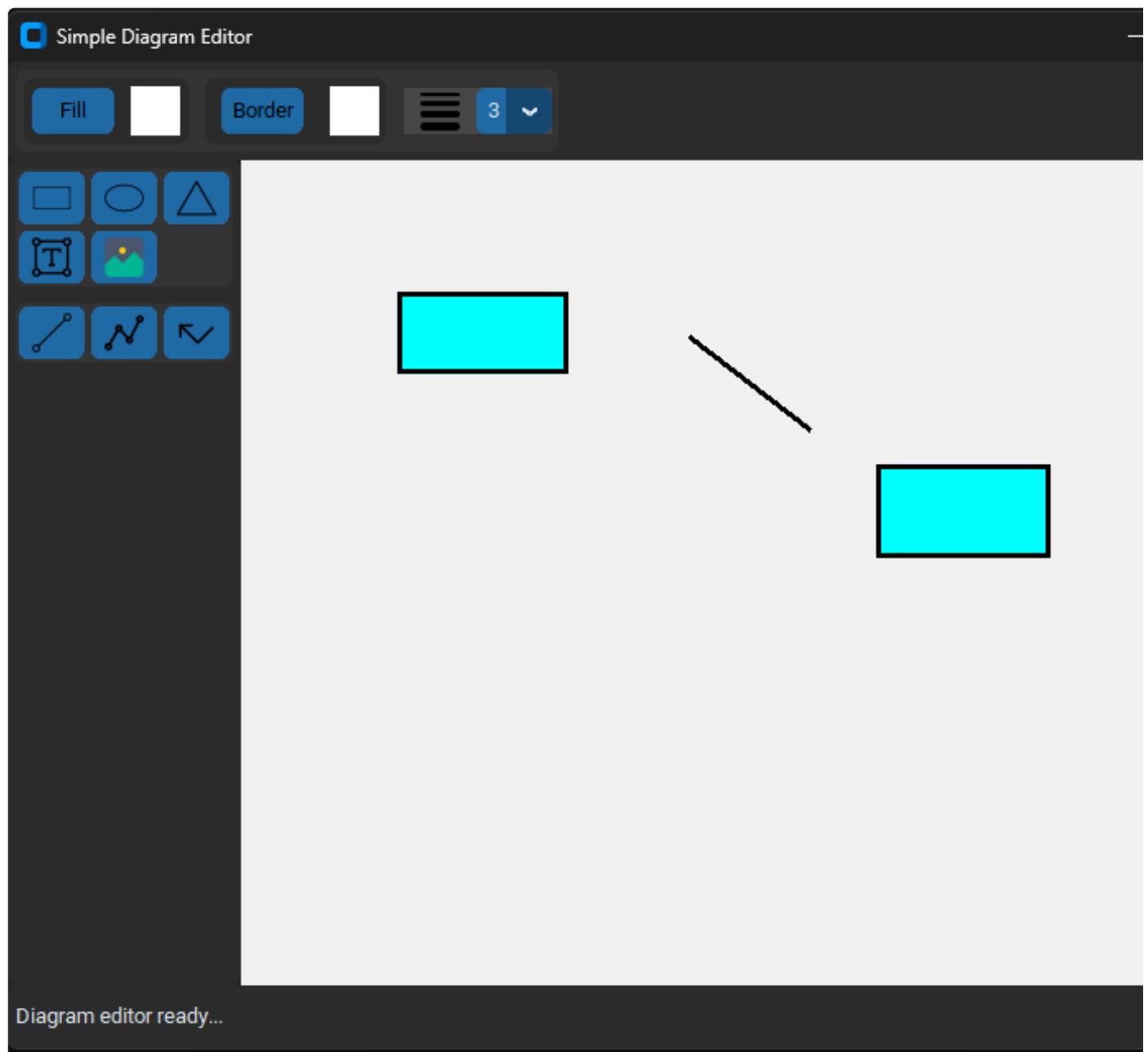
class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        shape_button_frame = ShapeButtonFrame(self, self.canvas)
        shape_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        line_button_frame = LineButtonFrame(self, self.canvas)
        line_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

```

Run the program, create two rectangles and a line and verify that there are no errors.



I like the custom button frames better than a vertical set of shape buttons. What do you think? Sometimes software programming is an art rather than a science.
Note that all the icons are available on GitHub.

Oval Class

We will model the Oval Class after the Rectangle Class.

oval.py

```
from Helper_Lib import Point  
from Shape_Lib.selector import Selector
```

```

from Shape_Lib.connector import Connector


class Oval:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None
        self.is_selected = False
        self.is_drawing = False
        self.sel_list = []
        self.conn_list = []
        self.line_list = []
        self.selector = None

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
        None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_oval(self.x1, self.y1, self.x2, self.y2,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:

```

```

        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    self.update_connectors()
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    self.move_connected_lines()

def create_selectors(self):
    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
    self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
    self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
    self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y1
    self.s2_id.update()

    self.s3_id.x, self.s3_id.y = self.x2, self.y2
    self.s3_id.update()

    self.s4_id.x, self.s4_id.y = self.x1, self.y2
    self.s4_id.update()

def create_connectors(self):
    """Create connectors here"""
    # Calculate the shape geometry
    w = self.x2 - self.x1
    h = self.y2 - self.y1

```

```

        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
        self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
        self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

def update_connectors(self):
    """Update the position of all connectors here"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id.x, self.c1_id.y = center.x, center.y
    self.c1_id.update()

    self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
    self.c2_id.update()

    self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
    self.c3_id.update()

    self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
    self.c4_id.update()

    self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
    self.c5_id.update()

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):

```

```

offset_x1, offset_y1, offset_x2, offset_y2 = offsets
if self.selector == "s1":
    x1 = event.x - offset_x1
    y1 = event.y - offset_y1
    self.x1, self.y1 = x1, y1
elif self.selector == "s2":
    x2 = event.x - offset_x2
    y1 = event.y - offset_y1
    self.x2, self.y1 = x2, y1
elif self.selector == "s3":
    x2 = event.x - offset_x2
    y2 = event.y - offset_y2
    self.x2, self.y2 = x2, y2
elif self.selector == "s4":
    x1 = event.x - offset_x1
    y2 = event.y - offset_y2
    self.x1, self.y2 = x1, y2

def check_selector_hit(self, x, y):
    for sel in self.sel_list:
        if sel.selector_hit_test(x, y):
            return sel
    return None

def check_connector_hit(self, x, y):
    for conn in self.conn_list:
        if conn.connector_hit_test(x, y):
            return conn
    return None

def move_connected_lines(self):
    for connection in self.line_list:
        for connector in self.conn_list:
            if connector == connection.connector_obj:
                # print(connector, connection.line_obj, "Match")
                if connection.line_end == "begin":
                    connection.line_obj.x1 = connector.x
                    connection.line_obj.y1 = connector.y
                elif connection.line_end == "end":
                    connection.line_obj.x2 = connector.x
                    connection.line_obj.y2 = connector.y

def __repr__(self):
    return ("Oval: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
str(self.y2) + ")\\nfill color: " + self.fill_color +

```

```
"\nborder_color: " + self.border_color +  
"\nborder_width: " + str(self.border_width))
```

Shape Base Class

Looking at the three shape classes: Rectangle, Oval, and `StraightLine` we can consider making a Shape Class as a base class with common attributes and methods. Create a new file in Shape_Lib called `shape.py`.

`shape.py`

```
class Shape:  
    def __init__(self, canvas, x1, y1, x2, y2):  
        self.canvas = canvas  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
  
        self.fill_color = "cyan"  
        self.border_color = "black"  
        self.border_width = 3  
  
        self.id = None  
        self.is_selected = False  
        self.is_drawing = False  
        self.sel_list = []  
        self.conn_list = []  
        self.line_list = []  
        self.selector = None  
  
    def check_selector_hit(self, x, y):  
        for sel in self.sel_list:  
            if sel.selector_hit_test(x, y):  
                return sel  
        return None  
  
    def check_connector_hit(self, x, y):  
        for conn in self.conn_list:  
            if conn.connector_hit_test(x, y):  
                return conn  
        return None
```

```

def move_connected_lines(self):
    for connection in self.line_list:
        for connector in self.conn_list:
            if connector == connection.connector_obj:
                # print(connector, connection.line_obj, "Match")
                if connection.line_end == "begin":
                    connection.line_obj.x1 = connector.x
                    connection.line_obj.y1 = connector.y
                elif connection.line_end == "end":
                    connection.line_obj.x2 = connector.x
                    connection.line_obj.y2 = connector.y

```

Rectangle Class derived from Shape Class

```

from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector

class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                         fill=self.fill_color,
                                         outline=self.border_color,
                                         width=self.border_width)

    def update(self):
        """Update the shape here"""

```

```

        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

        self.move_connected_lines()

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x2, self.y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = self.x1, self.y2

```

```

    self.s4_id.update()

def create_connectors(self):
    """Create connectors here"""
    # Calculate the shape geometry
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
    self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
    self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
    self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
    self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

    self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
    for c in self.conn_list:
        self.canvas.itemconfig(c.id, state='hidden')

def update_connectors(self):
    """Update the position of all connectors here"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id.x, self.c1_id.y = center.x, center.y
    self.c1_id.update()

    self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
    self.c2_id.update()

    self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
    self.c3_id.update()

    self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
    self.c4_id.update()

    self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
    self.c5_id.update()

```

```

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        self.x1, self.y2 = x1, y2

def __repr__(self):
    return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " + str(self.y2) + ")\\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

```

Run the program and verify that you can create a rectangle with no errors.

Refactored Oval Class

```

from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector

class Oval(Shape):

```

```

def __init__(self, canvas, x1, y1, x2, y2):
    super().__init__(canvas, x1, y1, x2, y2)

    self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
    self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
None, None, None, None

    self.create_shape()
    self.create_selectors()
    self.create_connectors()

def create_shape(self):
    """Create the shape here"""
    self.id = self.canvas.create_oval(self.x1, self.y1, self.x2, self.y2,
                                    fill=self.fill_color,
                                    outline=self.border_color,
                                    width=self.border_width)

def update(self):
    """Update the shape here"""
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, outline=self.border_color)
    self.canvas.itemconfig(self.id, width=self.border_width)

    self.update_selectors()
    if self.is_selected:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    self.update_connectors()
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    self.move_connected_lines()

def create_selectors(self):
    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)

```

```

        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y1
    self.s2_id.update()

    self.s3_id.x, self.s3_id.y = self.x2, self.y2
    self.s3_id.update()

    self.s4_id.x, self.s4_id.y = self.x1, self.y2
    self.s4_id.update()

def create_connectors(self):
    """Create connectors here"""
    # Calculate the shape geometry
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
    # Shape Center
    self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
    # Top Center
    self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
    # Right Center
    self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
    # Bottom Center
    self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
    # Left Center

    self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
    for c in self.conn_list:
        self.canvas.itemconfig(c.id, state='hidden')

def update_connectors(self):

```

```

"""Update the position of all connectors here"""
w = self.x2 - self.x1
h = self.y2 - self.y1
center = Point(self.x1+w/2, self.y1+h/2)

self.c1_id.x, self.c1_id.y = center.x, center.y
self.c1_id.update()

self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
self.c2_id.update()

self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
self.c3_id.update()

self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
self.c4_id.update()

self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
self.c5_id.update()

def rotate(self):
    w, h = self.x2 - self.x1, self.y2 - self.y1
    center = Point(self.x1 + w / 2, self.y1 + h / 2)
    self.x1, self.y1 = center.x - h/2, center.y - w/2
    self.x2, self.y2 = center.x + h/2, center.y + w/2

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "s1":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
    elif self.selector == "s2":
        x2 = event.x - offset_x2
        y1 = event.y - offset_y1
        self.x2, self.y1 = x2, y1
    elif self.selector == "s3":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "s4":
        x1 = event.x - offset_x1
        y2 = event.y - offset_y2
        self.x1, self.y2 = x1, y2

def __repr__(self):

```

```

        return ("Oval: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
            str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

```

Update the Shape Button Frame Class to create an oval.

```

import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image

class ShapeButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                           dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                           size=(24, 24))
        rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,
command=self.create_rect)
        rect_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(rect_button, msg="Rectangle")

        oval_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                           dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                           size=(24, 24))
        oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30,
command=self.create_oval) # Changed
        oval_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(oval_button, msg="Oval")

```

```

    tri_image = ctk.CTkImage(light_image=Image.open
        ("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
        dark_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
        size=(24, 24))
    tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30)
    tri_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
    ToolTip(tri_button, msg="Triangle")

    text_image = ctk.CTkImage(light_image=Image.open
        ("D:/EETools/SimpleDiagramEditor/icons/text.png"),
        dark_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/text.png"),
        size=(24, 24))
    text_button = ctk.CTkButton(self, text="", image=text_image, width=30)
    text_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
    ToolTip(text_button, msg="Text")

    pic_image = ctk.CTkImage(light_image=Image.open
        ("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
        dark_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
        size=(24, 24))
    pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30)
    pic_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
    ToolTip(pic_button, msg="Image")

    # Shape button handlers
def create_rect(self):
    self.canvas.mouse.current_shape = "rectangle"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def create_oval(self): # Added new method
    self.canvas.mouse.current_shape = "oval"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def hide_connectors(self):

```

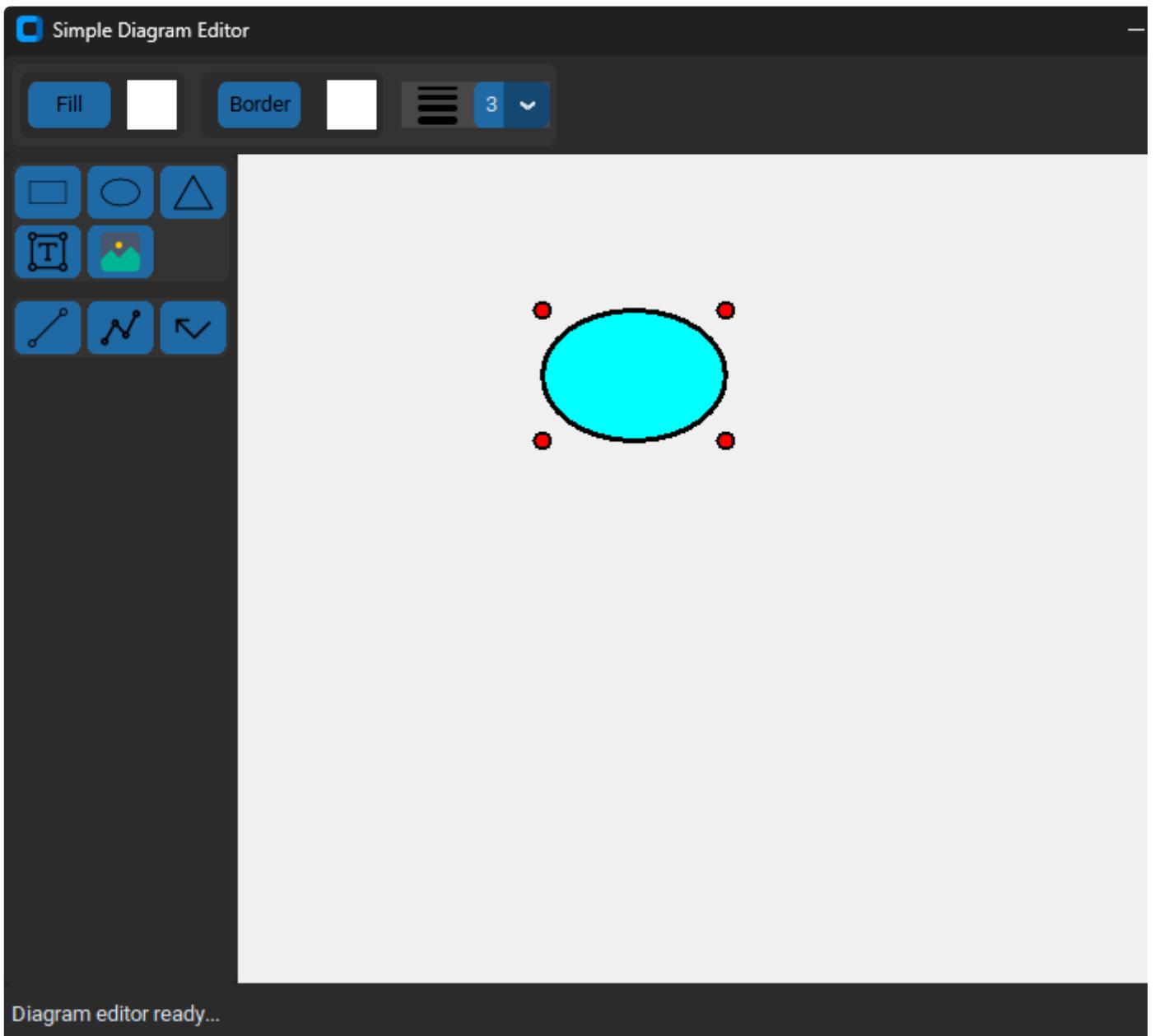
```
for s in self.canvas.shape_list:  
    s.is_drawing = False  
self.canvas.redraw_shapes()
```

Update the mouse class to draw an oval.

mouse.py modifications

```
from Helper_Lib.point import Point  
from Shape_Lib import Rectangle, Oval, StraightLine, Connection # Changed  
  
...  
  
def draw_left_down(self, event):  
    self.unselect_all()  
    self.start.x = event.x  
    self.start.y = event.y  
  
    if self.current_shape == "rectangle":  
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,  
self.start.y, self.start.x, self.start.y)  
    elif self.current_shape == "oval": # Added  
        self.current_shape_obj = Oval(self.canvas, self.start.x,  
self.start.y, self.start.x, self.start.y) # Added  
    elif self.current_shape == "straight":  
        self.current_shape_obj = StraightLine(self.canvas, self.start.x,  
self.start.y, self.start.x, self.start.y)  
        self.select_connector(self.current_shape_obj, "begin",  
self.start.x, self.start.y)  
  
    if self.current_shape_obj is not None:  
        self.canvas.shape_list.append(self.current_shape_obj)  
  
...
```

Run the program and verify that you can create an oval with all the same features as the rectangle.



Triangle Class

The Triangle Class shape is created with a polygon with three points defining an isosceles triangle.

triangle.py

```
from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector
```

```

class Triangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.points = []

        self.s1_id, self.s2_id, self.s3_id = None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id = None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape once!
        w = self.x2 - self.x1
        self.points = [self.x1 + w / 2, self.y1, self.x2, self.y2, self.x1,
        self.y2]
        self.id = self.canvas.create_polygon(self.points, fill=self.fill_color,
        outline=self.border_color,
        width=self.border_width)

    def update(self):
        w = self.x2 - self.x1
        self.points = [self.x1 + w / 2, self.y1, self.x2, self.y2, self.x1,
        self.y2]
        self.canvas.coords(self.id, self.points[0], self.points[1],
        self.points[2], self.points[3],
        self.points[4], self.points[5])
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:

```

```

        self.canvas.itemconfig(c.id, state='hidden')

    self.move_connected_lines()

def create_selectors(self):
    """Create four selectors at the corners here"""
    w = self.x2 - self.x1
    self.s1_id = Selector(self.canvas, "s1", self.x1 + w / 2, self.y1)
    self.s2_id = Selector(self.canvas, "s2", self.x2, self.y2)
    self.s3_id = Selector(self.canvas, "s3", self.x1, self.y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    """Update the position of all selectors here"""
    w = self.x2 - self.x1
    self.s1_id.x, self.s1_id.y = self.x1 + w / 2, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y2
    self.s2_id.update()

    self.s3_id.x, self.s3_id.y = self.x1, self.y2
    self.s3_id.update()

def create_connectors(self):
    """Create connectors here"""
    # Calculate the shape geometry
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
    # Shape Center
    self.c2_id = Connector(self.canvas, "c2", center.x + w/4, center.y)
    # Right Center
    self.c3_id = Connector(self.canvas, "c3", center.x - w/4, center.y)
    # Left Center
    self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
    # Bottom Center

    self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id]
    for c in self.conn_list:
        self.canvas.itemconfig(c.id, state='hidden')

```

```

def update_connectors(self):
    """Update the position of all connectors here"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    center = Point(self.x1+w/2, self.y1+h/2)

    self.c1_id.x, self.c1_id.y = center.x, center.y
    self.c1_id.update()

    self.c2_id.x, self.c2_id.y = center.x + w/4, center.y
    self.c2_id.update()

    self.c3_id.x, self.c3_id.y = center.x - w/4, center.y
    self.c3_id.update()

    self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
    self.c4_id.update()

def __repr__(self):
    return ("Triangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

```

Update the Shape Button Frame Class to create an oval.

```

import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image

class ShapeButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        rect_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),

```

```
        dark_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                size=(24, 24))
    rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,
command=self.create_rect)
    rect_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
    ToolTip(rect_button, msg="Rectangle")

    oval_image = ctk.CTkImage(light_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                dark_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                size=(24, 24))
    oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30,
command=self.create_oval)
    oval_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
    ToolTip(oval_button, msg="Oval")

    tri_image = ctk.CTkImage(light_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                dark_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                size=(24, 24))
    tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30,
command=self.create_tri) # Changed
    tri_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
    ToolTip(tri_button, msg="Triangle")

    text_image = ctk.CTkImage(light_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                dark_image=Image.open

        ("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                size=(24, 24))
    text_button = ctk.CTkButton(self, text="", image=text_image, width=30)
    text_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
    ToolTip(text_button, msg="Text")

    pic_image = ctk.CTkImage(light_image=Image.open
```

```

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
    dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
    size=(24, 24))
pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30)
pic_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
ToolTip(pic_button, msg="Image")

# Shape button handlers
def create_rect(self):
    self.canvas.mouse.current_shape = "rectangle"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def create_oval(self):
    self.canvas.mouse.current_shape = "oval"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def create_tri(self): # Added new method
    self.canvas.mouse.current_shape = "tri"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def hide_connectors(self):
    for s in self.canvas.shape_list:
        s.is_drawing = False
    self.canvas.redraw_shapes()

```

Modify the Mouse Class to draw a triangle.

```

from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, StraightLine, Connection # Changed

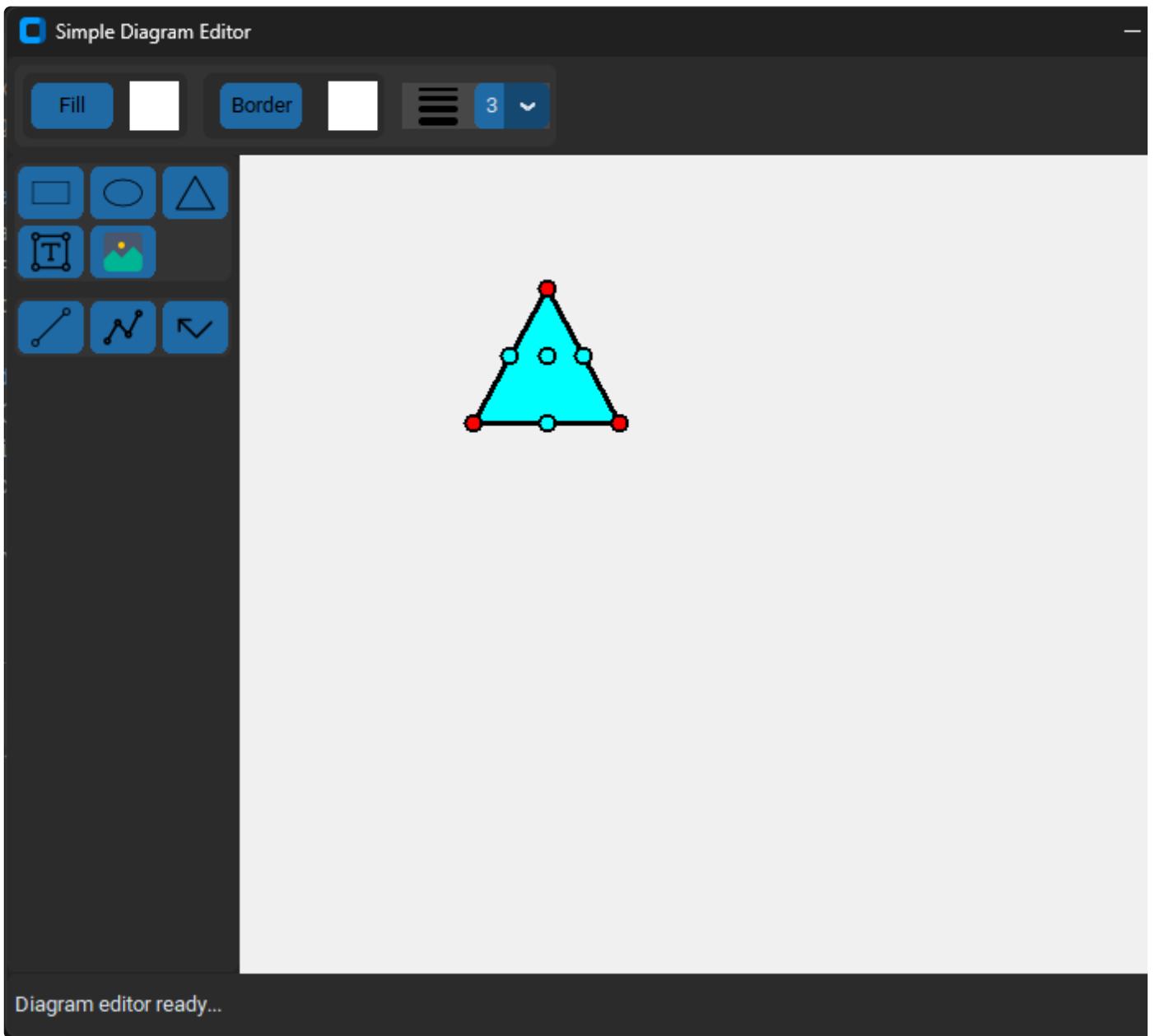
. . .

def draw_left_down(self, event):
    self.unselect_all()
    self.start.x = event.x
    self.start.y = event.y

```

```
if self.current_shape == "rectangle":  
    self.current_shape_obj = Rectangle(self.canvas, self.start.x,  
self.start.y, self.start.x, self.start.y)  
elif self.current_shape == "oval":  
    self.current_shape_obj = Oval(self.canvas, self.start.x,  
self.start.y, self.start.x, self.start.y)  
elif self.current_shape == "tri":  
    self.current_shape_obj = Triangle(self.canvas, self.start.x,  
self.start.y, self.start.x, self.start.y)  
elif self.current_shape == "straight":  
    self.current_shape_obj = StraightLine(self.canvas, self.start.x,  
self.start.y, self.start.x, self.start.y)  
    self.select_connector(self.current_shape_obj, "begin",  
self.start.x, self.start.y)  
  
if self.current_shape_obj is not None:  
    self.canvas.shape_list.append(self.current_shape_obj)  
  
. . .
```

Run the program and verify that you can draw a triangle with the same features as a rectangle.



Picture Class

The Picture Class will allow the user to add images from the file system to the diagram.
`picture.py`

```
from PIL import Image, ImageTk
import tkinter as tk

from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector
```

```

class Picture(Shape):
    def __init__(self, canvas, x1, y1, x2=0, y2=0):
        super().__init__(canvas, x1, y1, x2, y2)
        self.a_image = None
        self.ph_image = None
        self.filename = "D:/EETools/DiagramEditor/images/hamburger.png"
        self.angle = 0
        self.bbox = None
        self.type = "picture"

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id = None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape here"""
        self.a_image = Image.open(self.filename)
        self.a_image = self.a_image.rotate(self.angle)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags="pics")
        self.bbox = self.canvas.bbox(self.id)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1)

        self.a_image = Image.open(self.filename)
        self.a_image = self.a_image.rotate(self.angle)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.canvas.itemconfig(self.id, image=self.ph_image)

        self.bbox = self.canvas.bbox(self.id)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

```

```

        self.update_connectors()

    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    self.move_connected_lines()

def create_selectors(self):
    # Calculate position of selectors from current shape position
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]

    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "s1", x1, y1)
    self.s2_id = Selector(self.canvas, "s2", x2, y1)
    self.s3_id = Selector(self.canvas, "s3", x2, y2)
    self.s4_id = Selector(self.canvas, "s4", x1, y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    # Recalculate position of selectors from current shape position
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]

    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = x1, y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = x2, y1
    self.s2_id.update()

    self.s3_id.x, self.s3_id.y = x2, y2
    self.s3_id.update()

    self.s4_id.x, self.s4_id.y = x1, y2
    self.s4_id.update()

def create_connectors(self):
    """Create connectors here"""
    # Calculate the shape geometry
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1

```

```

        h = y2 - y1
        center = Point(x1+w/2, y1+h/2)

        self.c1_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
        self.c2_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
        self.c3_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
        self.c4_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

def update_connectors(self):
    """Update the position of all connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1+w/2, y1+h/2)

    self.c1_id.x, self.c1_id.y = center.x, center.y - h/2
    self.c1_id.update()

    self.c2_id.x, self.c2_id.y = center.x + w/2, center.y
    self.c2_id.update()

    self.c3_id.x, self.c3_id.y = center.x, center.y + h/2
    self.c3_id.update()

    self.c4_id.x, self.c4_id.y = center.x - w/2, center.y
    self.c4_id.update()

def rotate(self):
    """Calculate rotation angle"""
    self.angle += 90
    if self.angle > 270:
        self.angle = 0
    self.update()

def resize(self, offsets, event):
    pass

def __repr__(self):

```

```
        return ("Picture: x, y = " + "(" + str(self.x1) + ", " + str(self.y1) +  
               "\nimage filename = " + self.filename)
```

shape_button_frame.py modifications.

```
import customtkinter as ctk  
from tktooltip import ToolTip  
from PIL import Image # Added  
  
class ShapeButtonFrame(ctk.CTkFrame):  
    def __init__(self, parent, canvas):  
        super().__init__(parent)  
        self.parent = parent  
        self.canvas = canvas  
  
        # Add left frame widgets here  
        frame_name_label = ctk.CTkLabel(self, text="Shapes", font=("Helvetica",  
10), height=20)  
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,  
padx=2, pady=2)  
  
        rect_image = ctk.CTkImage(light_image=Image.open  
( "D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),  
                               dark_image=Image.open  
( "D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),  
                               size=(24, 24))  
        rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,  
command=self.create_rect)  
        rect_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)  
        ToolTip(rect_button, msg="Rectangle")  
  
        oval_image = ctk.CTkImage(light_image=Image.open  
( "D:/EETools/SimpleDiagramEditor/icons/oval.png"),  
                               dark_image=Image.open  
( "D:/EETools/SimpleDiagramEditor/icons/oval.png"),  
                               size=(24, 24))  
        oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30,  
command=self.create_oval)
```

```

oval_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
ToolTip(oval_button, msg="Oval")

tri_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
size=(24, 24))
tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30,
command=self.create_tri)
tri_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)
ToolTip(tri_button, msg="Triangle")

text_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/text.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
size=(24, 24))
text_button = ctk.CTkButton(self, text="", image=text_image, width=30,
command=self.create_text)
text_button.grid(row=2, column=0, sticky=ctk.W, padx=2, pady=2)
ToolTip(text_button, msg="Text")

pic_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
size=(24, 24))
pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30,
command=self.create_picture) # Changed
pic_button.grid(row=2, column=1, sticky=ctk.W, padx=2, pady=2)
ToolTip(pic_button, msg="Image")

# Shape button handlers
def create_rect(self):
    self.canvas.mouse.current_shape = "rectangle"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def create_oval(self):

```

```

        self.canvas.mouse.current_shape = "oval"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_tri(self):
        self.canvas.mouse.current_shape = "tri"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_text(self):
        self.canvas.mouse.current_shape = "text"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_picture(self): # Added new method
        self.canvas.mouse.current_shape = "pic"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def hide_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = False
        self.canvas.redraw_shapes()

```

mouse.py modifications.

```

from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture # Added Picture
from Shape_Lib import StraightLine, Connection

. . .

def draw_left_down(self, event):
    self.unselect_all()
    self.start.x = event.x
    self.start.y = event.y

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "oval":
        self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)

```

```

        elif self.current_shape == "tri":
            self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "text":
            self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "pic": # Added
            self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y) # Added
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)

    . . .

def select_hit_test(self, x, y):
    for s in self.canvas.shape_list:
        if isinstance(s, Text) or isinstance(s, Picture): # Changed to
check for Picture
            if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
                self.selected_shape = s
                s.is_selected = True
                self.canvas.redraw_shapes()
                return
        elif s.x1 <= x <= s.x2 and s.y1 <= y <= s.y2:
            self.selected_shape = s
            s.is_selected = True
            self.canvas.redraw_shapes()
            return

    # No shape hit - unselect all
    self.selected_shape = None
    self.unselect_all()

    . . .

```

canvas.py modifications

```

import customtkinter as ctk
from tkinter import filedialog # Added
from UI_Lib.mouse import Mouse
from Shape_Lib import Text

class Canvas(ctk.CTkCanvas):
    def __init__(self, master):
        super().__init__(master)
        self.shape_list = []
        self.mouse = Mouse(self)

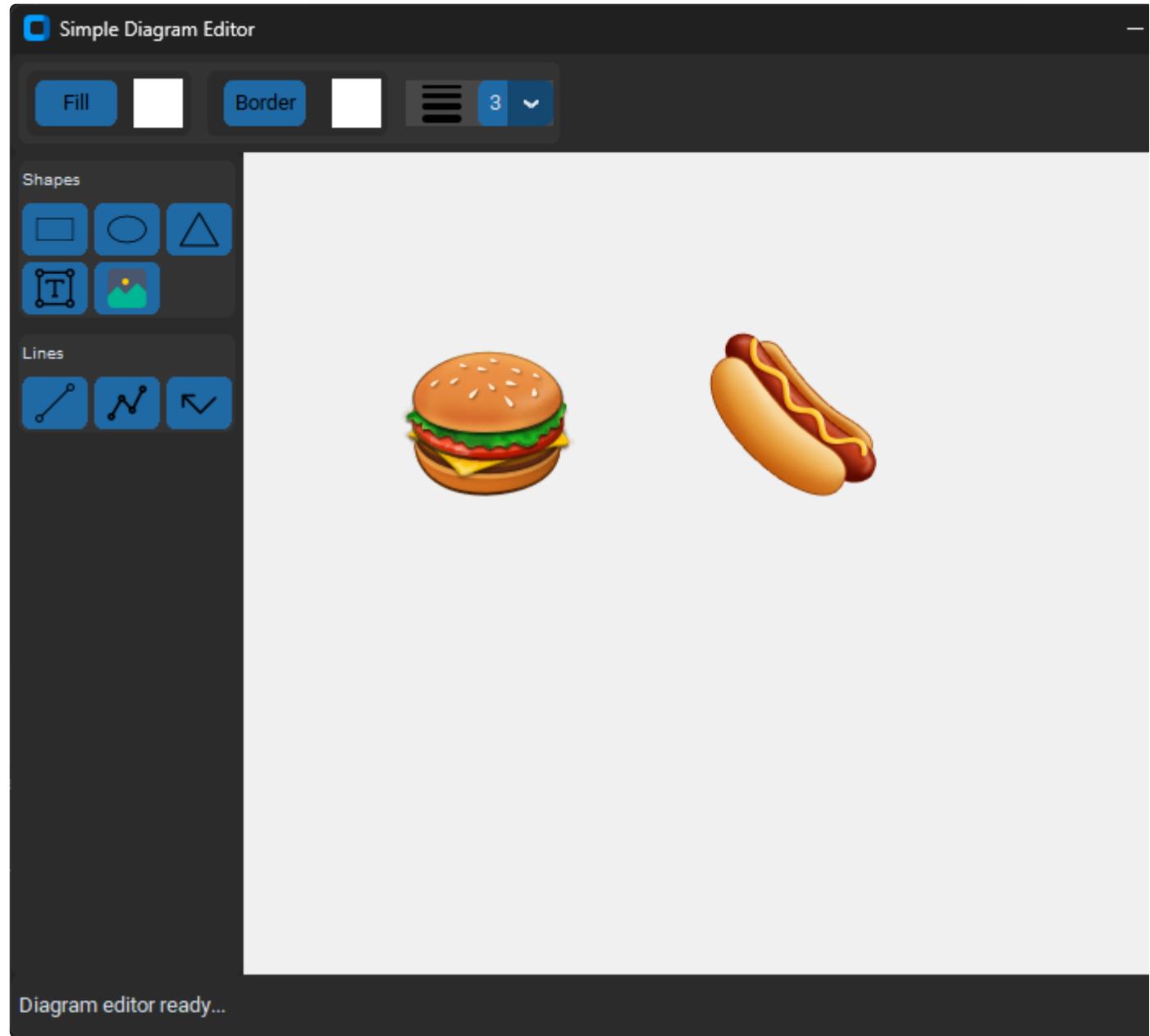
    def redraw_shapes(self):
        for s in self.shape_list:
            s.update()

    def edit_shape(self, _event):
        if self.mouse.selected_shape is not None:
            shape = self.mouse.selected_shape
            if self.gettags("current")[0] == "pics": # Added
                filename = filedialog.askopenfilename(initialdir="../images",
title="select a file", # Added
filetypes=(("png files",
 "*.png"), ("all file", "*.*")))) # Added
                shape.filename = filename # Added
                shape.update() # Added
            elif isinstance(shape, Text): # Changed
                dialog = ctk.CTkInputDialog(text="Enter new text", title="Edit
Text")
                shape.text = dialog.get_input()
                self.redraw_shapes()

```

Run the program, add two pictures of hamburgers (default), right click on one of the pictures and set it to a hot dog image. Verify that you can move, rotate, select, and connect

to the image.



Segment Line Class

The segment line has three line segments. The `canvas.create_line()` method can draw lines with multiple segments. We will add a feature to change the starting direction to horizontal or vertical.

`segment_line.py`

```
from Shape_Lib.selector import Selector
from Shape_Lib.shape import Shape
```

```

class SegmentLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.fill_color = "black"
        self.border_width = 3
        self.line_direction = "horizontal"
        self.segment_list = []

        self.seg1_id, self.seg2_id, self.seg3_id = None, None, None

        self.s1_id, self.s2_id = None, None

        self.create_shape()
        self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        if self.line_direction == "horizontal":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1 + w / 2, self.y1,
                                              self.x1 + w / 2, self.y1, self.x1 + w / 2, self.y2,
                                              self.x2, self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width)
        elif self.line_direction == "vertical":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1, self.y1 + h / 2,
                                              self.x1, self.y1 + h / 2, self.x2, self.y1 + h / 2,
                                              self.x2, self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width)

    def update(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        if self.line_direction == "horizontal":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1 + w / 2, self.y1,

```

```

                self.x1 + w / 2, self.y1, self.x1 + w / 2,
self.y2,
                self.x1 + w / 2, self.y2, self.x2, self.y2)
elif self.line_direction == "vertical":
    self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y1 + h
/ 2,
                self.x1, self.y1 + h / 2, self.x2, self.y1 + h /
2,
                self.x2, self.y1 + h / 2, self.x2, self.y2)
self.canvas.itemconfig(self.id, fill=self.fill_color)
self.canvas.itemconfig(self.id, width=self.border_width)

self.update_selectors()
if self.is_selected:
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='normal')
else:
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def create_selectors(self):
    """Create four selectors at the corners here"""
    self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
    self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

    self.sel_list = [self.s1_id, self.s2_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y2
    self.s2_id.update()

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1

```

```

        self.x1, self.y1 = x1, y1

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

line_button_frame.py modifications.

```

import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image

class LineButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Lines", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        straight_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                    dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                    size=(24, 24))
        straight_button = ctk.CTkButton(self, text="", image=straight_image,
width=30,
                                    command=self.create_straight_line)
        straight_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(straight_button, msg="Straight Line")

```

```

segment_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
size=(24, 24))
segment_button = ctk.CTkButton(self, text="", image=segment_image,
width=30, # Changed
command=self.create_segment_line) # Changed
segment_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
ToolTip(segment_button, msg="Segment Line")

elbow_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
size=(24, 24))
elbow_button = ctk.CTkButton(self, text="", image=elbow_image,
width=30)
elbow_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)
ToolTip(elbow_button, msg="Elbow Line")

# Shape button handlers
def create_straight_line(self):
    self.canvas.mouse.current_shape = "straight"
    self.show_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def create_segment_line(self): # Added new method
    self.canvas.mouse.current_shape = "segment"
    self.show_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def show_connectors(self):
    for s in self.canvas.shape_list:
        s.is_drawing = True
    self.canvas.redraw_shapes()

```

Shape_Lib/ __init__.py modifications

```

# Import shapes
from Shape_Lib.rectangle import Rectangle
from Shape_Lib.oval import Oval
from Shape_Lib.triangle import Triangle
from Shape_Lib.text import Text
from Shape_Lib.picture import Picture

# Import lines
from Shape_Lib.straight_line import StraightLine
from Shape_Lib.segment_line import SegmentLine # Added
from Shape_Lib.connection import Connection

```

mouse.py modifications

```

from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture
from Shape_Lib import StraightLine, SegmentLine, Connection # Changed

. . .

def draw_left_down(self, event):
    self.unselect_all()
    self.start.x = event.x
    self.start.y = event.y

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "oval":
        self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "tri":
        self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "text":
        self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "pic":
        self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "straight":
        self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)

```

```

        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
    elif self.current_shape == "segment": # Added
        self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y) # Added
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y) # Added

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)

    . . .

def draw_left_up(self, event):
    if isinstance(self.current_shape_obj, StraightLine) or
isinstance(self.current_shape_obj, SegmentLine): # Changed
        self.select_connector(self.current_shape_obj, "end", event.x,
event.y)

    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.move_bind_mouse_events()

    . . .

def select_connector(self, line_obj, line_end, x, y):
    for shape in self.canvas.shape_list:
        if not isinstance(shape, StraightLine) or not isinstance(shape,
SegmentLine): # Changed
            conn = shape.check_connector_hit(x, y)
            if conn:
                if line_end == "begin":
                    line_obj.x1, line_obj.y1 = conn.x, conn.y
                elif line_end == "end":
                    line_obj.x2, line_obj.y2 = conn.x, conn.y
                    a_conn = Connection(conn, line_obj, line_end)
                    shape.line_list.append(a_conn)
                    self.canvas.redraw_shapes()
    return

```

keyboard.py modifications.

```

class Keyboard:
    def __init__(self, parent, canvas):
        """Class to manage keyboard events"""
        self.parent = parent
        self.canvas = canvas
        self.selected_shape = None

        # Declare keyboard bindings
        self.parent.bind('<r>', self.rotate_shape)
        self.parent.bind('<h>', self.set_horizontal) # Added
        self.parent.bind('<v>', self.set_vertical) # Added

    def rotate_shape(self, _event):
        for s in self.canvas.shape_list:
            if s.is_selected:
                s.rotate()
        self.canvas.redraw_shapes()

    def set_horizontal(self, _event): # Added
        self.canvas.line_direction = "horizontal" # Added

    def set_vertical(self, _event): # Added
        self.canvas.line_direction = "vertical" # Added

```

canvas.py modifications.

```

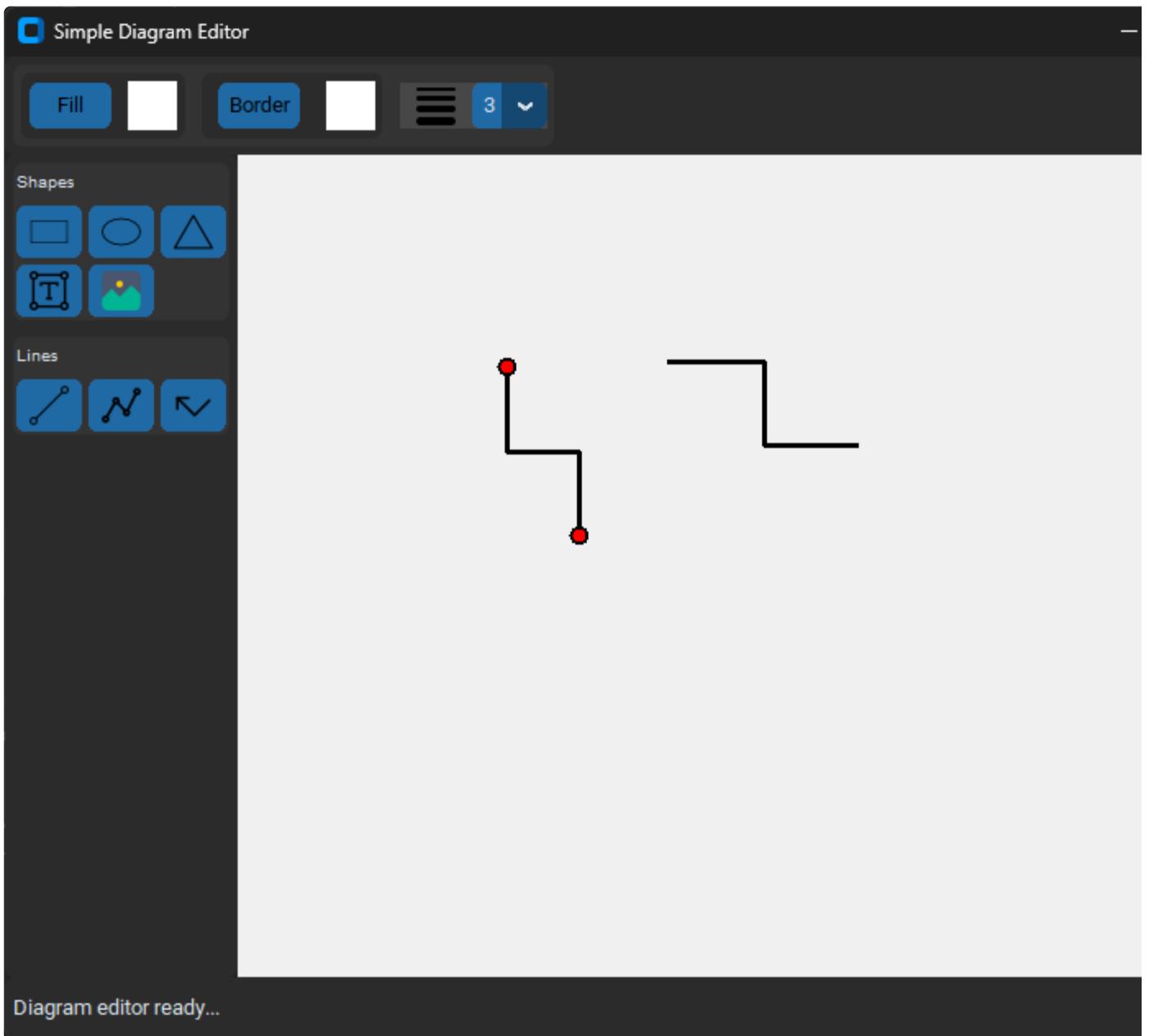
. . .

class Canvas(ctk.CTkCanvas):
    def __init__(self, master):
        super().__init__(master)
        self.shape_list = []
        self.mouse = Mouse(self)
        self.line_direction = "horizontal" # Added

. . .

```

Run the program, press the 'v' key, draw a segment line, press the 'h' key, draw another segment line.



Drawing segment lines is not as easy as it looks but we successfully programmed the new class.

Elbow Line Class

The Elbow Line Class is a simplified segment line with two segments in an L-shape or elbow-shape.

[elbow_line.py](#)

```
from Shape_Lib.selector import Selector  
from Shape_Lib.shape import Shape
```

```

class ElbowLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.fill_color = "black"
        self.border_width = 3
        self.line_direction = self.canvas.line_direction
        self.segment_list = []

        self(seg1_id, self(seg2_id, self(seg3_id = None, None, None

        self.s1_id, self.s2_id = None, None

        self.create_shape()
        self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        if self.line_direction == "horizontal":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x2,
self.y1,
                                         self.x2, self.y1, self.x2,
self.y2,
                                         fill=self.fill_color,
                                         width=self.border_width)
        elif self.line_direction == "vertical":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1,
self.y2,
                                         self.x1, self.y2, self.x2,
self.y2,
                                         fill=self.fill_color,
                                         width=self.border_width)

    def update(self):
        if self.line_direction == "horizontal":
            self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y1,
                               self.x2, self.y1, self.x2, self.y2)
        elif self.line_direction == "vertical":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y2,
                               self.x1, self.y2, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:

```

```

        self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)

```

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image

class LineButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add left frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Lines", font=("Helvetica", 10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W, padx=2, pady=2)

        straight_image = ctk.CTkImage(light_image=Image.open("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                       dark_image=Image.open("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                       size=(24, 24))
        straight_button = ctk.CTkButton(self, text="", image=straight_image, width=30,
                                         command=self.create_straight_line)
        straight_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(straight_button, msg="Straight Line")

        segment_image = ctk.CTkImage(light_image=Image.open("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                      dark_image=Image.open("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                      size=(24, 24))
        segment_button = ctk.CTkButton(self, text="", image=segment_image, width=30,
                                         command=self.create_segment_line)
        segment_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(segment_button, msg="Segment Line")

        elbow_image = ctk.CTkImage(light_image=Image.open("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
```

```

dark_image=Image.open("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
size=(24, 24))
elbow_button = ctk.CTkButton(self, text="", image=elbow_image,
width=30, # Changed
                                command=self.create_elbow_line) # Changed
elbow_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)
ToolTip(elbow_button, msg="Elbow Line")

# Shape button handlers
def create_straight_line(self):
    self.canvas.mouse.current_shape = "straight"
    self.show_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def create_segment_line(self):
    self.canvas.mouse.current_shape = "segment"
    self.show_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def create_elbow_line(self): # Added new method
    self.canvas.mouse.current_shape = "elbow"
    self.show_connectors()
    self.canvas.mouse.draw_bind_mouse_events()

def show_connectors(self):
    for s in self.canvas.shape_list:
        s.is_drawing = True
    self.canvas.redraw_shapes()

```

UI_Lib/__init__.py

```

# Import shapes
from Shape.Lib.rectangle import Rectangle
from Shape.Lib.oval import Oval
from Shape.Lib.triangle import Triangle
from Shape.Lib.text import Text
from Shape.Lib.picture import Picture

# Import lines
from Shape.Lib.straight_line import StraightLine

```

```
from Shape_Lib.segment_line import SegmentLine
from Shape_Lib.elbow_line import ElbowLine
from Shape_Lib.connection import Connection
```

mouse.py modifications

```
from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture
from Shape_Lib import StraightLine, SegmentLine, ElbowLine, Connection # Changed

. . .

def draw_left_down(self, event):
    self.unselect_all()
    self.start.x = event.x
    self.start.y = event.y

    if self.current_shape == "rectangle":
        self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "oval":
        self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "tri":
        self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "text":
        self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "pic":
        self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
    elif self.current_shape == "straight":
        self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
    elif self.current_shape == "segment":
        self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
```

```

        elif self.current_shape == "elbow": # Added
            self.current_shape_obj = ElbowLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y) # Added
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)

    . . .

def draw_left_up(self, event):
    if (isinstance(self.current_shape_obj, StraightLine) or
isinstance(self.current_shape_obj, SegmentLine) # Changed
        or isinstance(self.current_shape_obj, ElbowLine)): # Changed
        self.select_connector(self.current_shape_obj, "end", event.x,
event.y)

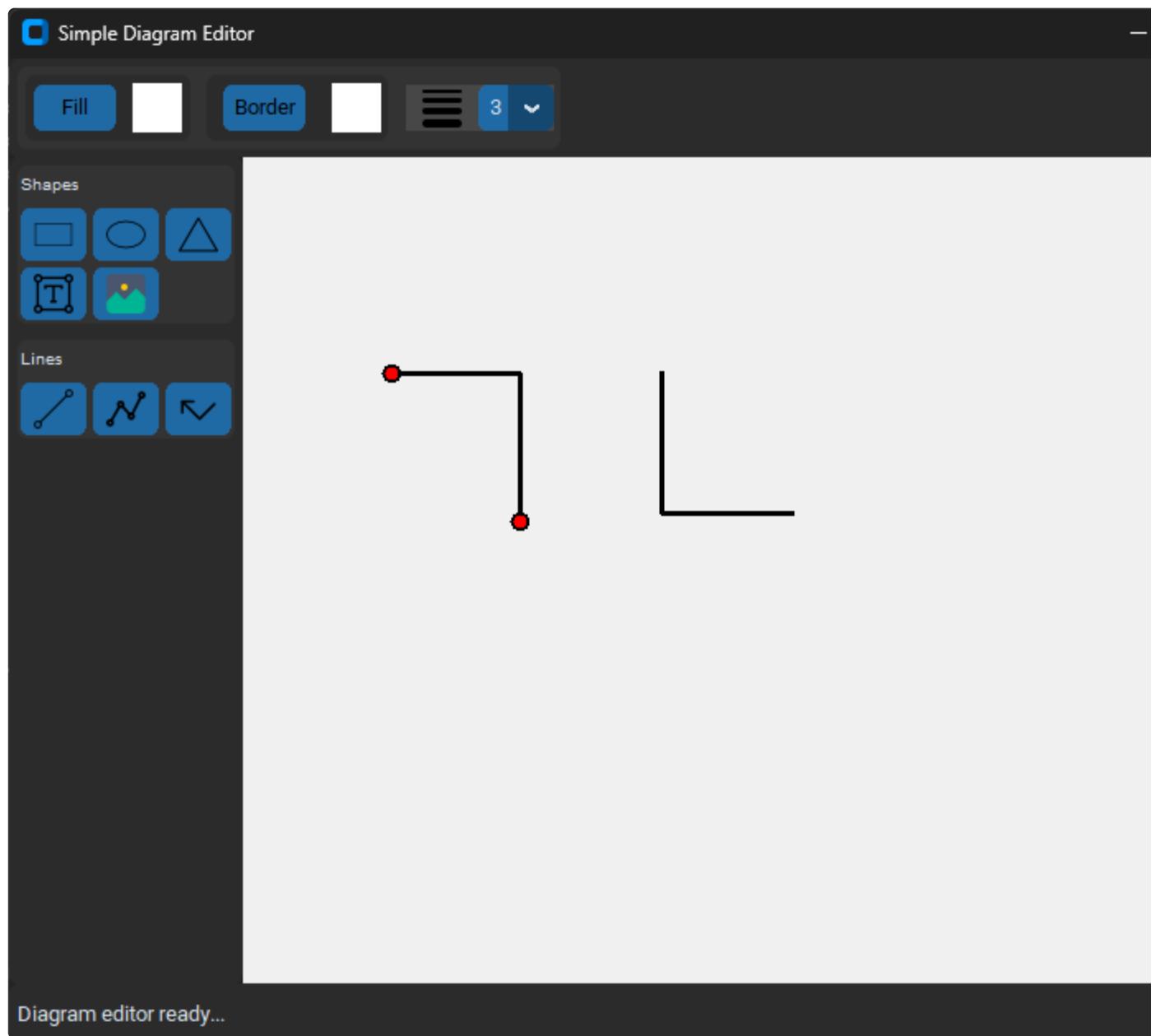
    self.current_shape = None
    self.current_shape_obj = None
    self.unbind_mouse_events()
    self.move_bind_mouse_events()

    . . .

def select_connector(self, line_obj, line_end, x, y):
    for shape in self.canvas.shape_list:
        if not isinstance(shape, StraightLine) or not isinstance(shape,
SegmentLine)\ # Changed
            or not isinstance(shape, ElbowLine): # Changed
            conn = shape.check_connector_hit(x, y)
            if conn:
                if line_end == "begin":
                    line_obj.x1, line_obj.y1 = conn.x, conn.y
                elif line_end == "end":
                    line_obj.x2, line_obj.y2 = conn.x, conn.y
                    a_conn = Connection(conn, line_obj, line_end)
                    shape.line_list.append(a_conn)
                    self.canvas.redraw_shapes()
    return

```

Run the program, draw a horizontal (default) elbow line, press the 'v' key, draw a vertical elbow line.



Grid Class

The Grid Class draws a background grid on the canvas at grid spacing intervals such as 10 px or 20 px. The `grid_snap` variable will set the snap size for shape draw, move, and resize. It also provide a Snap to Grid method that adjusts the provided x, y coordinates to the nearest snap point using the `round()` method.

`grid.py`

```
class Grid:  
    def __init__(self, canvas, grid_size):  
        self.canvas = canvas
```

```

        self.grid_size = grid_size
        self.grid_visible = True
        self.dash_list = None

        self.grid_snap = self.grid_size

    def draw(self):
        if self.grid_visible:
            w = self.canvas.winfo_width() # Get current width of canvas
            h = self.canvas.winfo_height() # Get current height of canvas

            # Creates all vertical lines at intervals of 100
            for i in range(0, w, self.grid_size):
                self.canvas.create_line([(i, 0), (i, h)], fill="#cccccc",
tags='grid_line')

            # Creates all horizontal lines at intervals of 100
            for i in range(0, h, self.grid_size):
                self.canvas.create_line([(0, i), (w, i)], fill="#cccccc",
tags='grid_line')

    def snap_to_grid(self, x, y):
        if self.grid_visible:
            x = round(x / self.grid_snap) * self.grid_snap
            y = round(y / self.grid_snap) * self.grid_snap
        return x, y

```

In the Main Application program we will add a binding to the `<configure>` event to detect a window resize. The window resize event handler will redraw the grid and all shapes.

`simple_diagram_editor.py`

```

import customtkinter as ctk
from UI_Lib import Canvas, TopFrame, LeftFrame, BottomFrame, Keyboard

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100") # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self, self.canvas)

```

```

        self.left_frame = LeftFrame(self, self.canvas)
        self.bottom_frame = BottomFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        self.keyboard = Keyboard(self, self.canvas)
        self.bind("<Configure>", self.on_window_resize) # Added

    def on_window_resize(self, _event): # Added
        self.canvas.redraw_shapes() # Added

if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()

```

Next we need to modify the Canvas Class redraw shapes method to draw the grid and lower its level (z-order) so it is always drawn in the background. We will also instantiate the Grid object in the Canvas Class initializer so it is drawn when the canvas is created.

canvas.py

```

import customtkinter as ctk
from tkinter import filedialog
from UI_Lib.mouse import Mouse
from Shape_Lib import Text
from Shape_Lib.grid import Grid # Added


class Canvas(ctk.CTkCanvas):
    def __init__(self, master):
        super().__init__(master)
        self.shape_list = []
        self.mouse = Mouse(self)
        self.line_direction = "horizontal"
        self.grid_size = 10 # Added
        self.grid = Grid(self, self.grid_size) # Added
        self.grid.draw() # Added

    def redraw_shapes(self):
        self.delete('grid_line') # Added

```

```

        self.grid.draw() # Added
        self.tag_lower('grid_line') # Added
        for s in self.shape_list:
            s.update()

    def edit_shape(self, _event):
        if self.mouse.selected_shape is not None:
            shape = self.mouse.selected_shape
            if self.gettags("current")[0] == "pics":
                filename = filedialog.askopenfilename(initialdir=..../images",
title="select a file",
filetypes=(( "png files",
".png"), ("all file", "*.*")))
                shape.filename = filename
                shape.update()
            elif isinstance(shape, Text):
                dialog = ctk.CTkInputDialog(text="Enter new text", title="Edit
Text")
                shape.text = dialog.get_input()
                self.redraw_shapes()

```

How do we snap shapes to the grid? We need to modify the Mouse Class to whenever coordinates are changed in the draw, move, and resize mouse handlers.

mouse.py

```

. . .

def move_left_down(self, event):
    x, y = event.x, event.y
    self.select_hit_test(x, y)
    if self.selected_shape:
        # print("Shape found: ", self.selected_shape)
        sel = self.selected_shape.check_selector_hit(x, y)
        if sel:
            self.selected_shape.selector = sel.name
            self.unbind_mouse_events()
            self.bind_resize_mouse_events()
            self.resize_left_down(event)
            return
        else:
            x1, y1 = self.selected_shape.x1, self.selected_shape.y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1) # Added
            x2, y2 = self.selected_shape.x2, self.selected_shape.y2

```

```

        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2) # Added
        self.offset1.x = x - x1
        self.offset1.y = y - y1
        self.offset2.x = x - x2
        self.offset2.y = y - y2

    def move_left_drag(self, event):
        if self.selected_shape:
            x = event.x - self.offset1.x
            y = event.y - self.offset1.y
            x, y = self.canvas.grid.snap_to_grid(x, y) # Added
            self.selected_shape.x1, self.selected_shape.y1 = x, y
            x = event.x - self.offset2.x
            y = event.y - self.offset2.y
            x, y = self.canvas.grid.snap_to_grid(x, y) # Added
            self.selected_shape.x2, self.selected_shape.y2 = x, y
            self.canvas.redraw_shapes()

    . . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y
        self.start.x, self.start.y =
        self.canvas.grid.snap_to_grid(self.start.x, self.start.y) # Added

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "oval":
            self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "tri":
            self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "text":
            self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "pic":
            self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",

```

```

        self.start.x, self.start.y)
    elif self.current_shape == "segment":
        self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
    elif self.current_shape == "elbow":
        self.current_shape_obj = ElbowLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

    if self.current_shape_obj is not None:
        self.canvas.shape_list.append(self.current_shape_obj)

def draw_left_drag(self, event):
    if self.current_shape_obj:
        shape = self.current_shape_obj
        x, y = event.x, event.y
        x, y = self.canvas.grid.snap_to_grid(x, y) # Added
        shape.x1, shape.y1 = self.start.x, self.start.y
        shape.x2, shape.y2 = x, y
        self.canvas.redraw_shapes()

    . . .

def resize_left_down(self, event):
    if self.selected_shape:
        shape = self.selected_shape
        x1, y1 = shape.x1, shape.y1
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1) # Added
        x2, y2 = shape.x2, shape.y2
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2) # Added
        self.offset1.x = event.x - x1
        self.offset1.y = event.y - y1
        self.offset2.x = event.x - x2
        self.offset2.y = event.y - y2

    . . .

```

Next we need to modify the `resize()` method in the Rectangle, Oval, and Triangle classes.

`rectangle.py`

```
    . . .  
  
def resize(self, offsets, event):  
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets  
    if self.selector == "s1":  
        x1 = event.x - offset_x1  
        y1 = event.y - offset_y1  
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1) # Added  
        self.x1, self.y1 = x1, y1  
    elif self.selector == "s2":  
        x2 = event.x - offset_x2  
        y1 = event.y - offset_y1  
        x2, y1 = self.canvas.grid.snap_to_grid(x2, y1) # Added  
        self.x2, self.y1 = x2, y1  
    elif self.selector == "s3":  
        x2 = event.x - offset_x2  
        y2 = event.y - offset_y2  
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2) # Added  
        self.x2, self.y2 = x2, y2  
    elif self.selector == "s4":  
        x1 = event.x - offset_x1  
        y2 = event.y - offset_y2  
        x1, y2 = self.canvas.grid.snap_to_grid(x1, y2) # Added  
        self.x1, self.y2 = x1, y2  
  
    . . .
```

oval.py

```
    . . .  
  
def resize(self, offsets, event):  
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets  
    if self.selector == "s1":  
        x1 = event.x - offset_x1  
        y1 = event.y - offset_y1  
        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1) # Added  
        self.x1, self.y1 = x1, y1  
    elif self.selector == "s2":  
        x2 = event.x - offset_x2  
        y1 = event.y - offset_y1  
        x2, y1 = self.canvas.grid.snap_to_grid(x2, y1) # Added  
        self.x2, self.y1 = x2, y1  
    elif self.selector == "s3":  
        . . .
```

```
x2 = event.x - offset_x2
y2 = event.y - offset_y2
x2, y2 = self.canvas.grid.snap_to_grid(x2, y2) # Added
self.x2, self.y2 = x2, y2

elif self.selector == "s4":
    x1 = event.x - offset_x1
    y2 = event.y - offset_y2
    x1, y2 = self.canvas.grid.snap_to_grid(x1, y2) # Added
    self.x1, self.y2 = x1, y2

.
.
```

Oops, we forgot to add a Triangle Class resize and rotate methods. Lets add them now and with `snap_to_grid()` method calls where needed.

triangle.py

```

def update(self):
    w = self.x2 - self.x1
    self.rotation_points()
    self.canvas.coords(self.id, self.points[0], self.points[1],
self.points[2], self.points[3],
                           self.points[4], self.points[5])
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, outline=self.border_color)
    self.canvas.itemconfig(self.id, width=self.border_width)

    self.update_selectors()
    if self.is_selected:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    self.update_connectors()
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    self.move_connected_lines()

def create_selectors(self):
    """Create four selectors at the corners here"""
    w = self.x2 - self.x1
    self.s1_id = Selector(self.canvas, "s1", self.x1 + w / 2, self.y1)
    self.s2_id = Selector(self.canvas, "s2", self.x2, self.y2)
    self.s3_id = Selector(self.canvas, "s3", self.x1, self.y2)

    self.sel_list = [self.s1_id, self.s2_id, self.s3_id]
    for s in self.sel_list:
        self.canvas.itemconfig(s.id, state='hidden')

def update_selectors(self):
    """Update the position of all selectors here"""
    x1, y1, x2, y2, x3, y3 = self.points
    w = x2 - x1
    self.s1_id.x, self.s1_id.y = x1, y1
    self.s1_id.update()

```

```

        self.s2_id.x, self.s2_id.y = x2, y2
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = x3, y3
        self.s3_id.update()

    def create_connectors(self):
        """Create connectors here"""
        # Calculate the shape geometry
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h / 2)
# Right Center
        self.c3_id = Connector(self.canvas, "c3", center.x + w / 4, center.y)
# Left Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h / 2)
# Bottom Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    def update_connectors(self): # Modified method
        """Update the position of all connectors here"""
        x1, y1, x2, y2, x3, y3 = self.points

        if self.angle == 0 or self.angle == 180:
            w = x2 - x3
            h = y2 - y1
            center = Point(x1 + w / 2, y1 + h / 2)
            self.c1_id.x, self.c1_id.y = x1, y1+h/2
            self.c1_id.update()
            self.c2_id.x, self.c2_id.y = x1 - w/4, center.y
            self.c2_id.update()
            self.c3_id.x, self.c3_id.y = x1 + w/4, center.y
            self.c3_id.update()
            self.c4_id.x, self.c4_id.y = x1, y2
            self.c4_id.update()
        elif self.angle == 90 or self.angle == 270:
            w = x2 - x1

```

```

        h = y2 - y3
        center = Point(x1 + w / 2, y1)
        self.c1_id.x, self.c1_id.y = x1 + w/2, y1
        self.c1_id.update()
        self.c2_id.x, self.c2_id.y = center.x, center.y - h/4
        self.c2_id.update()
        self.c3_id.x, self.c3_id.y = center.x, center.y + h/4
        self.c3_id.update()
        self.c4_id.x, self.c4_id.y = x2, y1
        self.c4_id.update()

    def rotate(self): # Added method
        # Calculate rotation angle
        self.angle += 90
        if self.angle > 270:
            self.angle = 0

    def rotation_points(self): # Added method
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        if self.angle == 0:
            self.points = [self.x1 + w / 2, self.y1, self.x2, self.y2, self.x1,
self.y2]
        elif self.angle == 90:
            self.points = [self.x1, self.y1 + h/2, self.x2, self.y1, self.x2,
self.y2]
        elif self.angle == 180:
            self.points = [self.x1 + w/2, self.y2, self.x1, self.y1, self.x2,
self.y1]
        elif self.angle == 270:
            self.points = [self.x2, self.y1 + h/2, self.x1, self.y2, self.x1,
self.y1]

    def resize(self, offsets, event): # Added method
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
            self.x2, self.y2 = x2, y2
        elif self.selector == "s3":
```

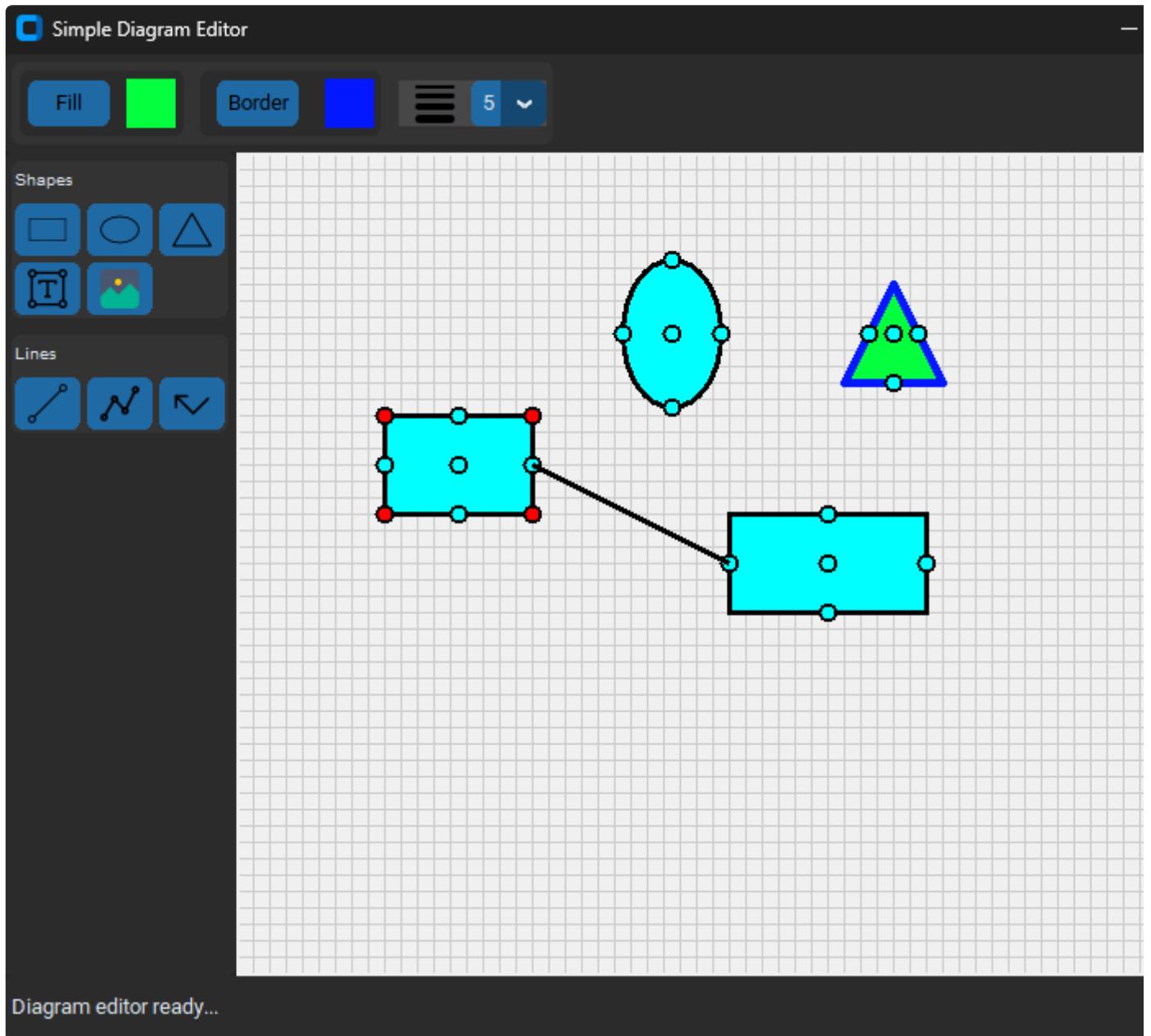
```

x1 = event.x - offset_x1
y2 = event.y - offset_y2
x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)
self.x1, self.y2 = x1, y2

def __repr__(self):
    return ("Triangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
str(self.y2) + ")\nfill_color: " + self.fill_color +
"\nborder_color: " + self.border_color +
"\nborder_width: " + str(self.border_width))

```

Run the program and draw rectangles, ovals, triangles and lines and verify that they snap to the grid spacing. Also verify that the triangle will rotate and resize.



Top Frame Class Update

Next we will add three menus to the Top Frame:

- File Menu
- Settings Menu
- Help Menu

File Menu Frame Class

file_menu_frame.py

```
import customtkinter as ctk
from tkinter import filedialog as fd
import json
from PIL import Image

from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture
from Shape_Lib import StraightLine, SegmentLine, ElbowLine


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas
        self.obj_type_dict = {'rectangle': Rectangle,
                             'oval': Oval,
                             'triangle': Triangle,
                             'text': Text,
                             'straight': StraightLine,
                             'segment': SegmentLine,
                             'elbow': ElbowLine,
                             'picture': Picture}

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
                               command=self.new_diagram)
        new_btn.pack(pady=5)
```

```

        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
command=window.destroy)
        exit_btn.pack(pady=5)

    my_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/DiagramEditor/icons/hamburger_menu.png"),
dark_image=Image.open
("D:/EETools/DiagramEditor/icons/hamburger_menu.png"),
size=(24, 24))

    button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
    button.pack(side=ctk.LEFT, padx=5, pady=10)

def new_diagram(self):
    self.save_diagram()
    self.canvas.delete("all")
    self.canvas.shape_list = []

def load_diagram(self):
    try:
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
        with open(f) as file:
            obj_dict = json.load(file)
            for obj_type, attributes in obj_dict.items():
                obj = self.obj_type_dict[obj_type.split()[0]](self.canvas,
attributes[0], attributes[1],
attributes[2], attributes[3])
                self.canvas.shape_list.append(obj)
                self.canvas.redraw_shapes()

    except FileNotFoundError:
        with open('untitled.canvas', 'w') as file:
            pass

```

```

        self.canvas.shape_list = []

    def save_diagram(self):
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
        with open(f, 'w') as file:
            obj_dict = {f'{obj.type} {id}': (obj.x1, obj.y1, obj.x2, obj.y2)
for id, obj in
                enumerate(self.canvas.shape_list)}
            json.dump(obj_dict, file)

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

```

Settings Frame Class

settings_frame.py

```

import customtkinter as ctk
from PIL import Image

class SettingsFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

    def grid_switch_event():
        if canvas.grid.grid_visible:
            canvas.grid.grid_visible = False
            self.canvas.mouse.move_snap = 1
            self.canvas.mouse.resize_snap = 1
            self.canvas.mouse.rotate_snap = 90

```

```

        else:
            canvas.grid.grid_visible = True
            self.canvas.mouse.move_snap = 10
            self.canvas.mouse.resize_snap = 10
            self.canvas.mouse.rotate_snap = 90
            self.canvas.redraw_shapes()

switch_var = ctk.StringVar(value="on")
switch = ctk.CTkSwitch(self.menu_frame, text="Grid",
command=grid_switch_event,
variable=switch_var, onvalue="on",
offvalue="off")
switch.pack(padx=5, pady=5)

grid_size_label = ctk.CTkLabel(self.menu_frame, text="Grid Size", font=\
("Helvetica", 10), height=20)
grid_size_label.pack(padx=5, pady=5, anchor="w")

def optionmenu_callback(choice):
    self.canvas.grid.grid_size = int(choice)
    self.canvas.redraw_shapes()

optionmenu = ctk.CTkOptionMenu(self.menu_frame, values=[ "5", "10",
"20", "30", "40", "50"],
command=optionmenu_callback)
optionmenu.pack(padx=5, pady=5)
optionmenu.set("10")

grid_snap_label = ctk.CTkLabel(self.menu_frame, text="Snap Size", font=\
("Helvetica", 10), height=20)
grid_snap_label.pack(padx=5, pady=5, anchor="w")

def snap_option_callback(choice):
    if choice == "Grid Size":
        self.canvas.grid.grid_snap = self.canvas.grid.grid_size
    else:
        self.canvas.grid.grid_snap = int(choice)
    self.canvas.redraw_shapes()

snap_option = ctk.CTkOptionMenu(self.menu_frame, values=[ "Grid Size",
"5", "10", "20", "30", "40", "50"],
command=snap_option_callback)
snap_option.pack(padx=5, pady=5)
snap_option.set("Grid Size")

self.appearance_mode_label = ctk.CTkLabel(self.menu_frame,

```

```

text="Appearance Mode:", anchor="w")
    self.appearance_mode_label.pack(padx=5, pady=5)
    self.appearance_mode_optionmenu = ctk.CTkOptionMenu(self.menu_frame,
                                                       values=
["Light", "Dark", "System"],

command=self.change_appearance_mode_event)
    self.appearance_mode_optionmenu.pack(padx=5, pady=5)
    self.appearance_mode_optionmenu.set("Dark")

my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/DiagramEditor/icons/settings.png"),
                           dark_image=Image.open

("D:/EETools/DiagramEditor/icons/settings.png"),
                           size=(24, 24))

button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
button.pack(side=ctk.LEFT, padx=5, pady=10)

def show_menu(self):
    if not self.menu_on:
        self.menu_frame.place(x=15, y=60)
        self.menu_frame.tkraise()
        self.menu_on = True
    else:
        self.menu_frame.place_forget()
        self.menu_on = False

def change_appearance_mode_event(self, new_appearance_mode: str):
    ctk.set_appearance_mode(new_appearance_mode)

```

Help Frame Class

help_frame.py

```

import customtkinter as ctk
from tkinter import messagebox
from PIL import Image

```

```
class HelpFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.window = window
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        about_image = ctk.CTkImage(light_image=Image.open
            ("D:/EETools/SimpleDiagramEditor/icons/about.png"),
            dark_image=Image.open
            ("D:/EETools/SimpleDiagramEditor/icons/about.png"),
            size=(24, 24))

        about_button = ctk.CTkButton(self.menu_frame, text="About",
            image=about_image, width=30,
            command=self.show_about_dialog)
        about_button.pack(side=ctk.TOP, padx=5, pady=5)

        my_image = ctk.CTkImage(light_image=Image.open
            ("D:/EETools/SimpleDiagramEditor/icons/help.png"),
            dark_image=Image.open
            ("D:/EETools/SimpleDiagramEditor/icons/help.png"),
            size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
            command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            menu_pos_x = self.canvas.winfo_width()
            self.menu_frame.place(x=menu_pos_x + 50, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

    def show_about_dialog(self):
```

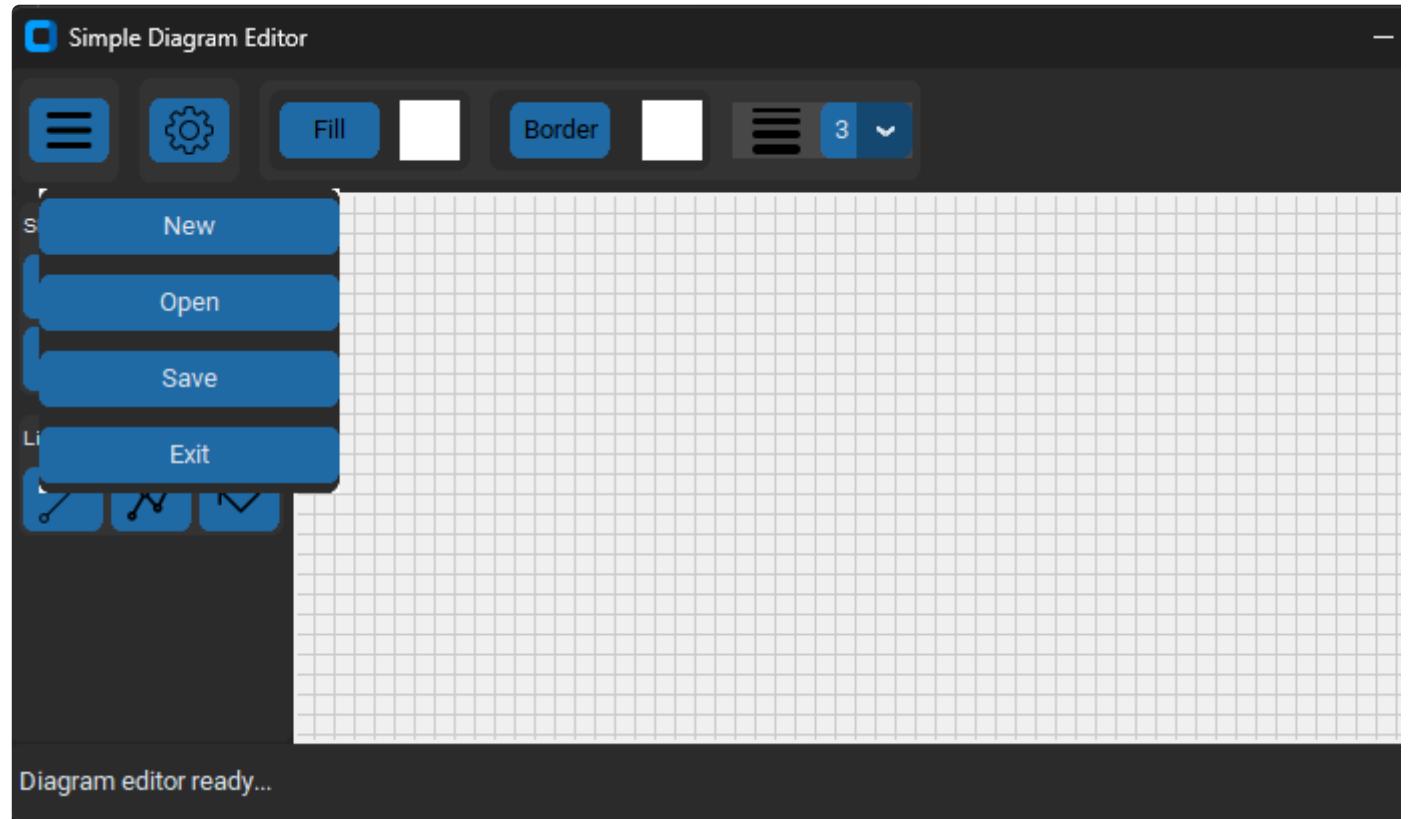
```
messagebox.showinfo("About Diagram Editor", "Diagram Editor v0.1\n" +  
"Author: Rick A. Crist\n" + "2023")
```

Top Frame Class

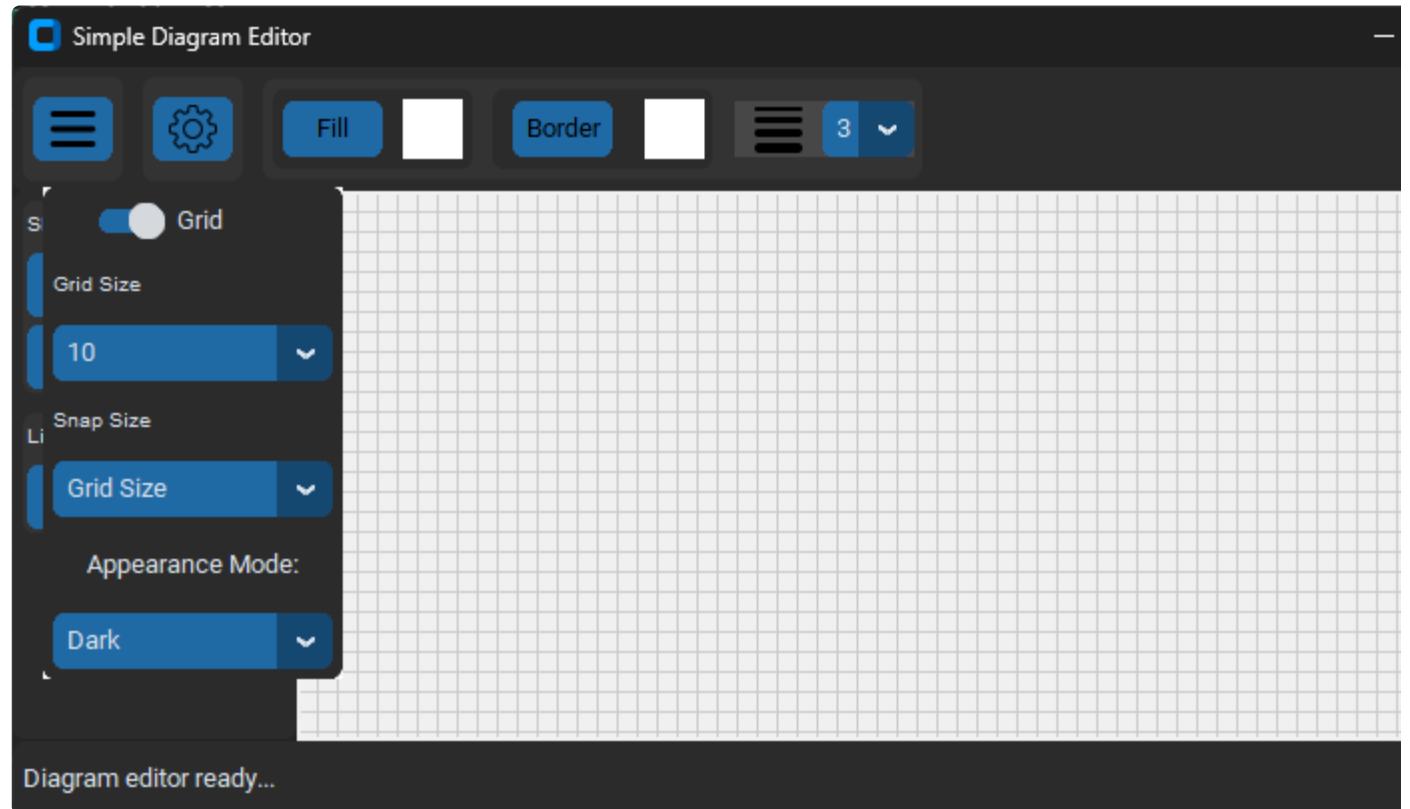
top_frame.py

```
import customtkinter as ctk  
from UI_Lib.shape_appearance_frame import ShapeAppearanceFrame  
from UI_Lib.file_menu_frame import FileMenuFrame  
from UI_Lib.settings_frame import SettingsFrame  
from UI_Lib.help_frame import HelpFrame  
  
class TopFrame(ctk.CTkFrame):  
    def __init__(self, parent, canvas):  
        super().__init__(parent)  
        self.parent = parent  
        self.canvas = canvas  
  
        # Add Top Frame widget here  
        file_frame = FileMenuFrame(self.parent, self, self.canvas)  
        file_frame.pack(side=ctk.LEFT, padx=5, pady=5)  
  
        settings_frame = SettingsFrame(self.parent, self, self.canvas)  
        settings_frame.pack(side=ctk.LEFT, padx=5, pady=5)  
  
        shape_frame = ShapeAppearanceFrame(self, self.canvas)  
        shape_frame.pack(side=ctk.LEFT, padx=5, pady=5)  
  
        help_frame = HelpFrame(self.parent, self, self.canvas)  
        help_frame.pack(side=ctk.RIGHT, padx=5, pady=5)
```

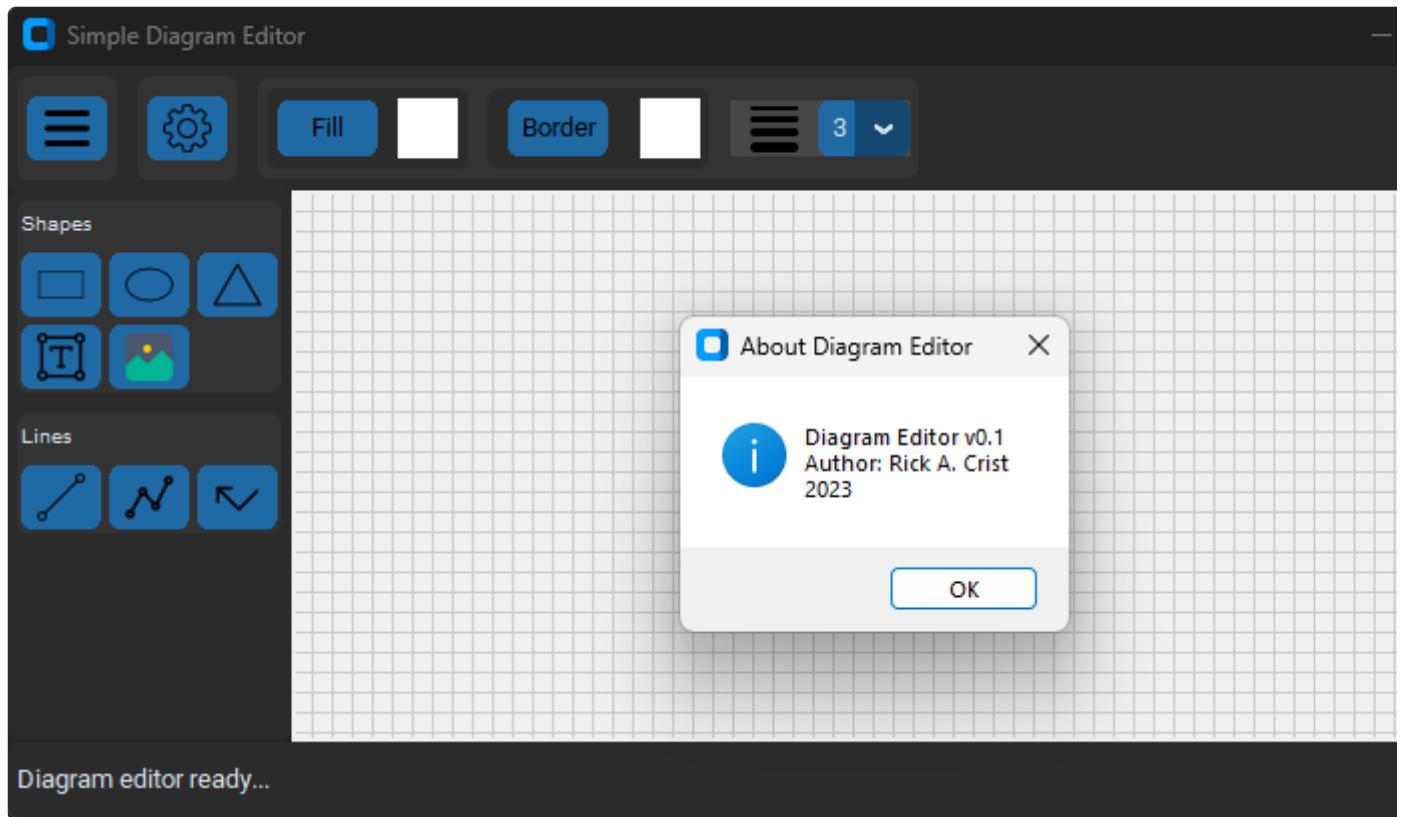
File Menu



Settings Menu



Help Menu



Status Messages

Lets go through the button handlers and send a message to the status message bar at the bottom of the screen.

Here is a list of button handlers:

- Shape Button Frame - 5 buttons
- Line Button Frame - 3 buttons

First, we need to refactor the Bottom Frame Class to add an update() method and make the text an attribute of the class.

bottom_frame.py

```
import customtkinter as ctk

class BottomFrame(ctk.CTkFrame):
    def __init__(self, parent):
        super().__init__(parent)
        self.text = "Diagram editor ready..." # Added
```

```
    self.message = ctk.CTkLabel(self, text=self.text) # Changed
    self.message.pack(side=ctk.LEFT, padx=5, pady=5)

def update(self): # Added
    self.message.configure(text=self.text) # Added
```

Here is an example from the Rectangle Button in the Shape Button Frame

```
def create_rect(self):
    self.canvas.mouse.current_shape = "rectangle"
    self.hide_connectors()
    self.canvas.mouse.draw_bind_mouse_events()
    self.parent.parent.bottom_frame.text = "Drawing Rectangle" # Added
    self.parent.parent.bottom_frame.update() # Added
```

Summary

This is the end of the Simple Diagram Editor application. We have come a long ways since starting this application development and due to its complexity, it is an advanced python project. It is also important for the future projects which all require a diagram capability to draw circuits and schematics.

You can model a circuit simulator as a Diagram Editor with a simulation capability. The next project is a Digital Circuit Simulator so lets proceed.

Chapter 6 - Digital Circuit Simulator

Design, Features, and Specifications

Digital logic circuits are currently the most common form of electronic circuits used in computers, cell phones, tablets, cars, digital radios, and most other electronic applications. Digital circuits consist of "logic" gates, memory, and clock circuits that when combined perform a high level function such as a microprocessor or digital radio. At first glance creating a digital simulator in Python seems rather trivial since the programming language has conditional statements and boolean variables such as `and`, `or`, `not`, `True`, and `False`. The challenge for this project is to design a real-time clock system that can operate efficiently and achieve perhaps greater than 100 Hz operation on a desktop computer.

Approach:

- ✓ Modify the Diagram Editor App to create the Digital Circuit Simulator
- ✓ KISS - Keep it simple, silly
- ✓ DRY - Don't repeat yourself
- ✓ SOC - Separation of concerns
- ✓ User-interface
 - ✓ TopFrame class
 - ✓ File menu frame
 - ✓ Settings menu frame
 - ✓ LED settings frame
 - ✓ Rotation button
 - ✓ Help menu frame
 - ✓ Left Frame Class with Circuit Component Menu
 - ✓ Canvas class
 - ✓ Mouse class
- ✓ Circuit components
 - ✓ AND gate
 - ✓ OR gate
 - ✓ NAND gate
 - ✓ NOR gate
 - ✓ NOT gate

- ✓ XOR gate
- ✓ XNOR gate
- ✓ Wire Class
- ✓ Clock class
- ✓ Grid class
- ✓ Switch class
- ✓ LED class
 - ✓ LED size options: large or small
 - ✓ LED color options: red, yellow, blue, green
- ✓ Text Class
- ✓ Analysis
 - ✓ Combinational Logic Simulation
 - ✓ Sequential Logic Simulation
 - ✓ Circuit traversal algorithm
- ✓ Counter Circuit Simulation
 - ✓ 74LS161 Synchronous 4-Bit Counter
 - ✓ 74LS273 Octal D Flip-Flop with Clear
 - ✓ 28C16 16K (2K x 8) Parallel EEPROMs
 - ✓ 7-Segment Display

Key Technologies Needed:

- ✓ File save & load in json format - multiple lists? - Component list and connection list

Project Setup

Language: Python 3.11

IDE: PyCharm 2023.2.1 (Community Edition)

Project directory: D:/EETools/DigitalSimulator

Graphics library: CustomTkinter (<https://customtkinter.tomschimansky.com/>)

External libraries:

- ✓ pip install customtkinter
- ✓ python.exe -m pip install --upgrade pip
- ✓ pip install ctkcolorpicker
- ✓ pip install tkinter-tooltip

- ✓ pip install pyInstaller - Create .exe file
- ✓ Add images and icons directories to the project.

Digital Components

Reference:

- "Digital Computer Electronics", A. P. Malvino, Ph. D. and J. A. Brown, Glencoe/McGraw-Hill, 3rd Edition, 1999, [Internet Archive PDF](#)

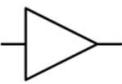
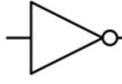
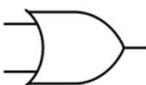
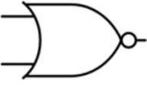
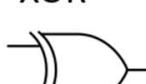
Buffer 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">Input</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> </tr> </tbody> </table>	Input	Output	0	0	1	1	Inverter 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">Input</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> </tr> </tbody> </table>	Input	Output	0	1	1	0																		
Input	Output																																
0	0																																
1	1																																
Input	Output																																
0	1																																
1	0																																
AND 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">A</th> <th style="padding: 2px;">B</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> </tr> </tbody> </table>	A	B	Output	0	0	0	1	0	0	0	1	0	1	1	1	NAND 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">A</th> <th style="padding: 2px;">B</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> </tr> </tbody> </table>	A	B	Output	0	0	1	1	0	1	0	1	1	1	1	0
A	B	Output																															
0	0	0																															
1	0	0																															
0	1	0																															
1	1	1																															
A	B	Output																															
0	0	1																															
1	0	1																															
0	1	1																															
1	1	0																															
OR 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">A</th> <th style="padding: 2px;">B</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> </tr> </tbody> </table>	A	B	Output	0	0	0	1	0	1	0	1	1	1	1	1	NOR 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">A</th> <th style="padding: 2px;">B</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> </tr> </tbody> </table>	A	B	Output	0	0	1	1	0	0	0	1	0	1	1	0
A	B	Output																															
0	0	0																															
1	0	1																															
0	1	1																															
1	1	1																															
A	B	Output																															
0	0	1																															
1	0	0																															
0	1	0																															
1	1	0																															
XOR 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">A</th> <th style="padding: 2px;">B</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> </tr> </tbody> </table>	A	B	Output	0	0	0	1	0	1	0	1	1	1	1	0	XNOR 	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px;">A</th> <th style="padding: 2px;">B</th> <th style="padding: 2px;">Output</th> </tr> </thead> <tbody> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">0</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> </tr> </tbody> </table>	A	B	Output	0	0	1	1	0	0	0	1	0	1	1	1
A	B	Output																															
0	0	0																															
1	0	1																															
0	1	1																															
1	1	0																															
A	B	Output																															
0	0	1																															
1	0	0																															
0	1	0																															
1	1	1																															

Image courtesy of [Global Science Network](#)

Digital logic gates can easily be represented in Python where inputs are True or False. The output is True or False.

```
output = input1 and input2 # Model for AND gate
```

Unfortunately, the canvas shape methods cannot draw a closed curved polygon. We will need to use image files for the graphical representation of the logic gates. We will use a set

of image files courtesy of [Logix](#).

Clock Component

For synchronous circuit simulation, we need a real-time clock simulator that generates a continuous set of pulses that toggle between 1 and 0 or True and False. We will use the threading library to run a continuous clock signal on a separate thread.

```
import threading
import time

state = False

def background_calculation():
    # set the time
    time.sleep(1)

    # Toggle and print state
    global state
    state = not state
    print(state)
    background_calculation()

def main():
    thread = threading.Thread(target=background_calculation)
    thread.start()

if __name__ == '__main__':
    main()
```

Console Output

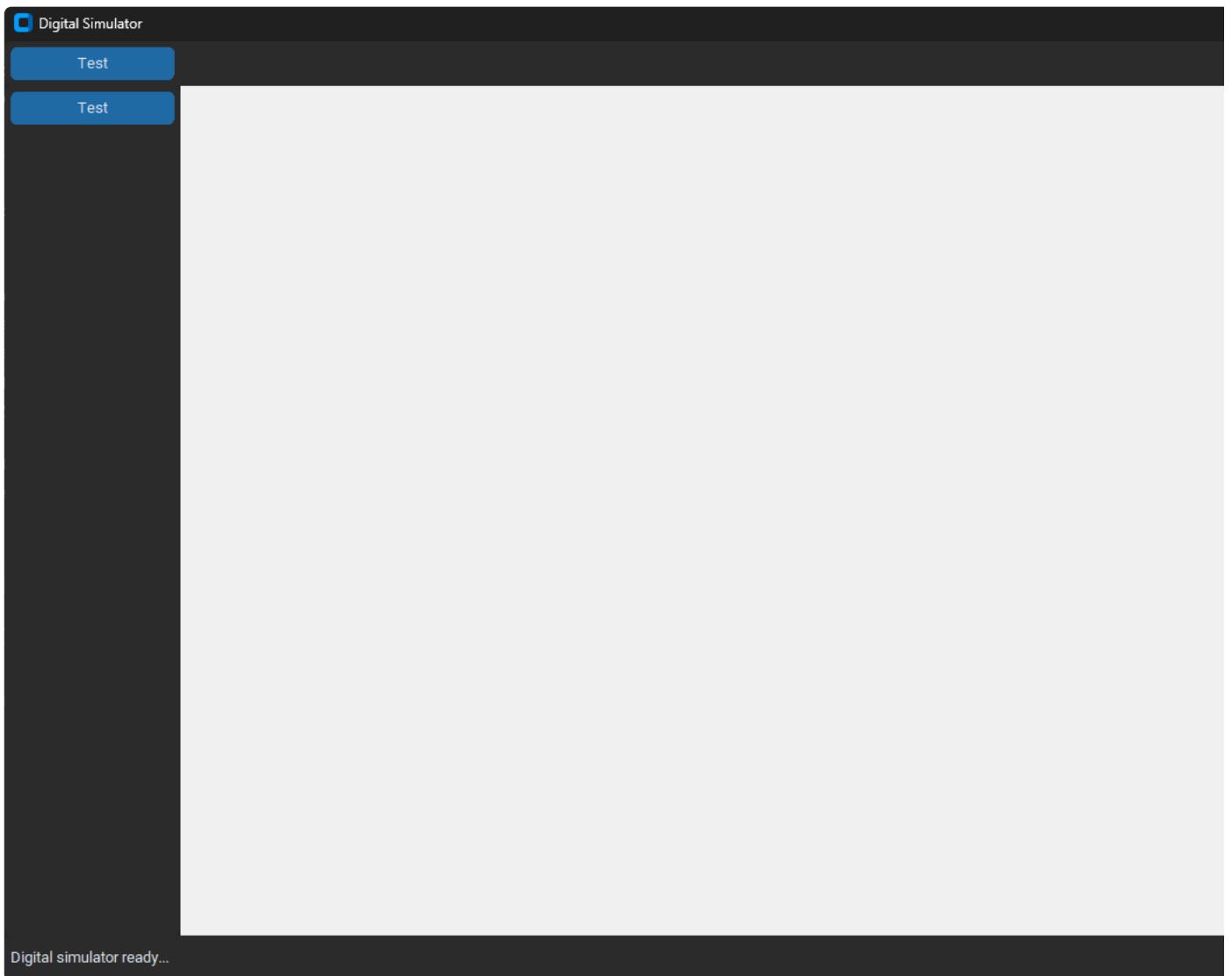
```
True
False
True
False
True
```

```
False  
True  
False  
True  
False  
True
```

Main User Interface

The initial user interface design is shown in the image below. It consists of a CustomTkinter window with a canvas widget and three custom frame widgets. Custom widgets will allow us to create a modular UI design so that the main file class is small and manageable.





digital_simulator.py

```
import customtkinter as ctk

ctk.set_appearance_mode("System") # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue") # Themes: "blue" (standard), "green",
"dark-blue"

class DigitalSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""

    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Digital Simulator")
```

```

        self.canvas = ctk.CTkCanvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add widgets to frames to cause the frames to auto-size
        top_frame_button = ctk.CTkButton(self.top_frame, text="Test")
        top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

        left_frame_button = ctk.CTkButton(self.left_frame, text="Test")
        left_frame_button.pack(side=ctk.TOP, padx=5, pady=5)

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready..."))
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()

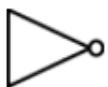
```

Digital Gate Design

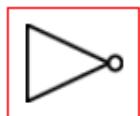
Objectives:

- Create a Not Gate class
- Gate is an image file
- Add selector - shown when the gate is selected
- Add connectors - shown when drawing wires
- Draw gate on canvas from a left frame button

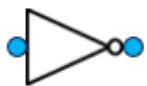
Not Gate Decorators



Unselected

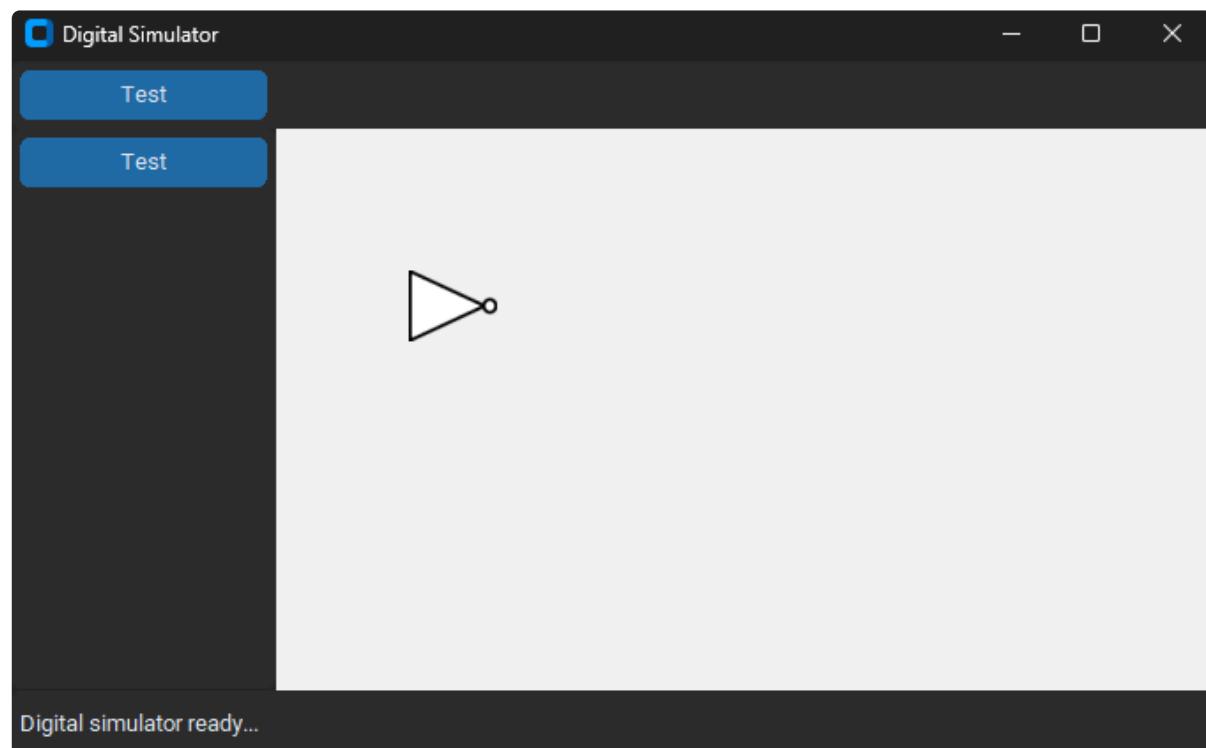


Selector



Connectors

Not Gate Class



Comp_Lib/not_gate.py

```
import tkinter as tk
from pathlib import Path
from PIL import Image, ImageTk
```

```

class NotGate:
    def __init__(self, canvas, x1, y1):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0

        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)

    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
                                         image=self.ph_image, tags='gate')

```

`digital_simulator.py`

```

import customtkinter as ctk
from Comp_Lib import NotGate # Added Not gate import here

. . .

bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")
bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

# Create test canvas widget here
self.gate = NotGate(self.canvas, 100, 100) # Created not gate object
here

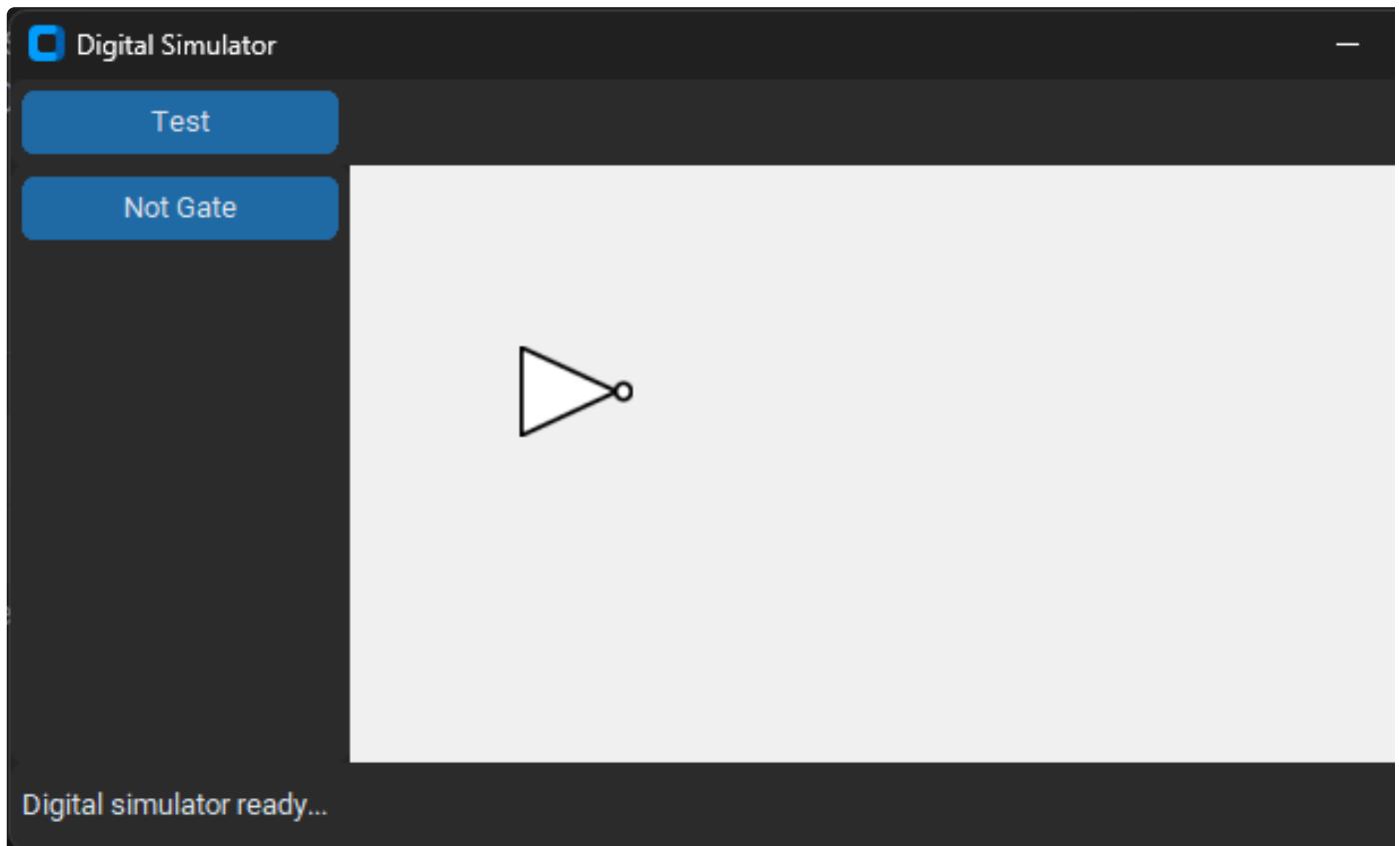
if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""

```

```
app = DigitalSimulatorApp()
app.mainloop()
```

Note: The not gate object must be created using `self.gate` not `gate` or it will be deleted by the Python garbage collector and will not be shown on the screen when the program is run.

Create a Not gate from a left frame button



`digital_simulator.py`

```
...
left_frame_button = ctk.CTkButton(self.left_frame, text="Not Gate",
command=self.create_not_gate) # Added call to button handler
left_frame_button.pack(side=ctk.TOP, padx=5, pady=5)
...
```

```

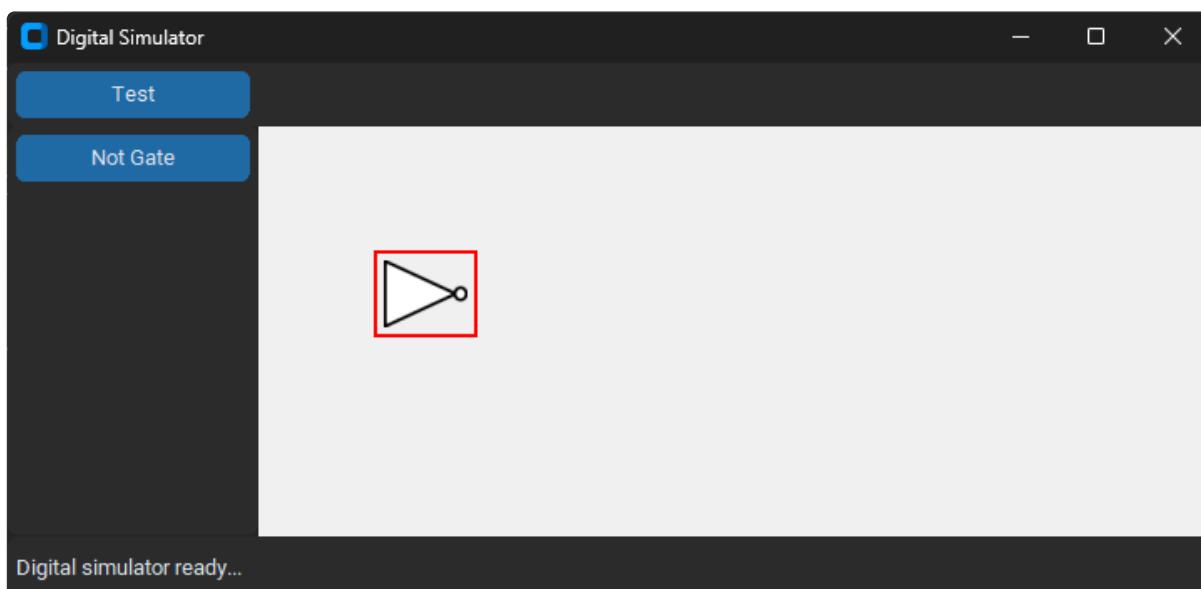
# Create test canvas widget here
    # self.gate = NotGate(self.canvas, 100, 100)
    self.gate = None # Variable to store the gate object

def create_not_gate(self): # Added new method
    self.gate = NotGate(self.canvas, 100, 100)

if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()

```

Add selector to Not Gate class



Comp_Lib/not_gate.py

```

import tkinter as tk
from pathlib import Path
from PIL import Image, ImageTk

class NotGate:
    def __init__(self, canvas, x1, y1):

```

```

    self.canvas = canvas
    self.x1 = x1
    self.y1 = y1

    self.id = None
    self.sel_id = None # Added variable for selector id
    self.a_image = None
    self.ph_image = None
    self.bbox = None
    self.angle = 0

    self.is_selected = True # Added boolean variable for gate selection

    self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
    self.create_image(self.filename)
    self.update_bbox() # Added call to update bbox
    self.create_selector() # Added call to create the selector

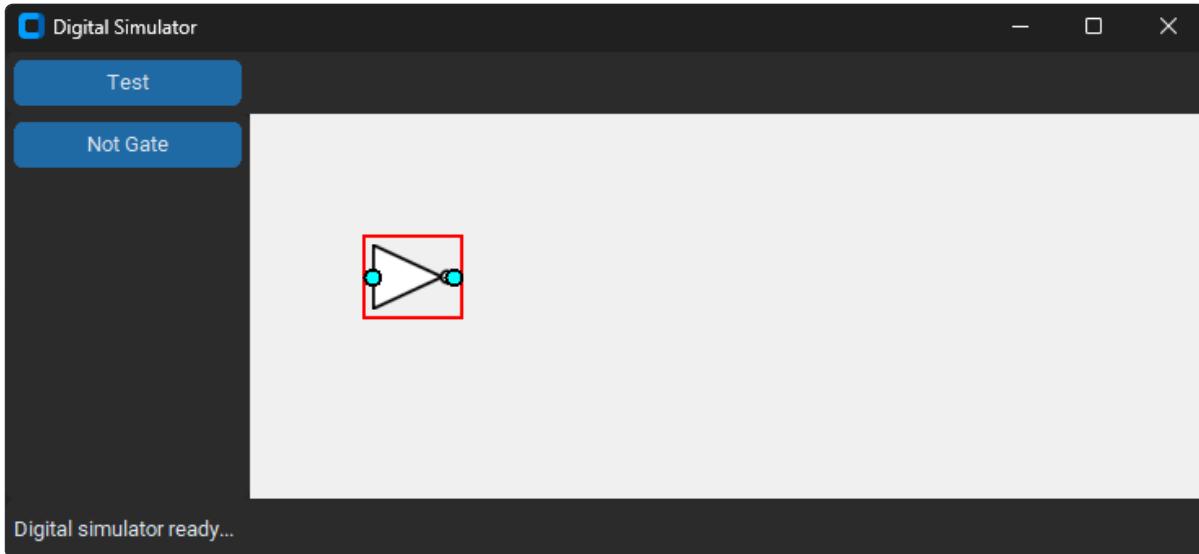
def create_image(self, filename):
    """Initial component image creation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True)
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

def create_selector(self): # Added method to create selector
    """Create the red rectangle selector and check to see if the gate is
selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)

def update_bbox(self): # Added method to update the bounding box (bbox) to
the gate's current location
    """Update the bounding box to get current gate coordinates"""
    self.bbox = self.canvas.bbox(self.id)

```

Not Gate Connectors



New Connector Class

Connector class features:

- Given the canvas and coordinates the class initializer creates a cyan oval at the x, y coordinates
 - `update()` method updates the connector position using the `canvas.coords()` method
 - `set_position()` method that sets the connector coordinates to provided x, y coordinates
 - Connector hit test method to determine if the provided x, y coordinates are within the bounds of the connector

Wire_Lib/connector.py

```

    self.radius)

    self.create_connector()

def create_connector(self):
    # Create the connector here
    points = [self.x - self.radius, self.y - self.radius, self.x +
    self.radius, self.y + self.radius]
    self.id = self.canvas.create_oval(points, fill="cyan", outline="black",
width=2, tags='connector')

def update(self):
    """Update the connector here"""
    self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
    self.radius,
                                         self.x + self.radius, self.y +
    self.radius)
    points = [self.x - self.radius, self.y - self.radius, self.x +
    self.radius, self.y + self.radius]
    self.canvas.coords(self.id, points)

def set_pos(self, x, y):
    """Set the connector position here"""
    self.x = x
    self.y = y

def connector_hit_test(self, x, y):
    """Connector hit test"""
    if self.x1 <= x <= self.x2 and self.y1 <= y <= self.y2:
        return True
    else:
        return False

def __repr__(self):
    return ("Connector: " + self.name + " (" + str(self.x1) + ", " +
str(self.y1) + ")" +
        " (" + str(self.x2) + ", " + str(self.y2) + ")")

```

New Point Class

Point class features:

- Create points with .x and .y notation

- Eliminates the need for [0] and [1] on point lists or `_x` or `_y` notation on variable names

Helper_Lib/point.py

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Comp_Lib/not_gate.py

```
import tkinter as tk
from pathlib import Path
from PIL import Image, ImageTk

from Wire_Lib import Connector # Added import for the Connector class
from Helper_Lib import Point # Added import for the new Point Class

. . .

    self.angle = 0

    self.is_selected = True
    self.is_drawing = True # Added boolean that is set when a wire is
being draw

    self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
    self.create_image(self.filename)
    self.update_bbox()
    self.create_selector()

    # Create 2 connectors
    self.in1_id, self.out1_id = None, None # Added
    self.conn_list = [] # Add a connector list
    self.create_connectors() # Added call to create connectors

. . .
```

```

def create_connectors(self): # Added new method to create connectors
    """Create connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    self.out1_id = Connector(self.canvas, "out1", center.x + w / 2,
center.y) # Out1
    self.in1_id = Connector(self.canvas, "in1", center.x - w / 2, center.y)
# In1
    self.conn_list = [self.out1_id, self.in1_id]

```

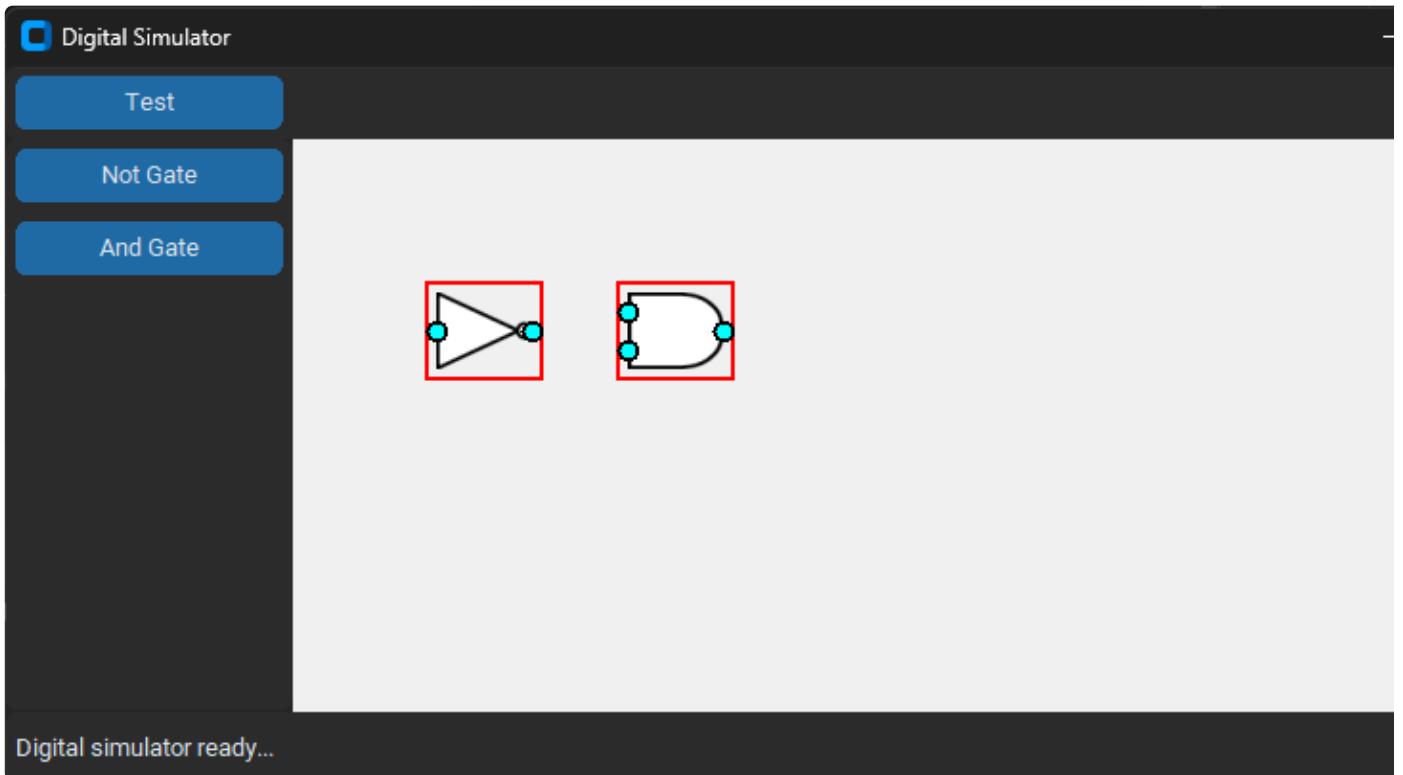
Notes:

- Connectors are positioned at the input and output points on the Not gate symbol
- Positions are calculated relative to the center of the Not gate dimensions
- Connectors have names that will be used later for wire connections

And Gate Class

Objectives:

- Create a new And Gate Class
- Create a new Comp Base Class for And Gate and Not Gate (for DRY)
- Create common selector and connectors for 2-input gates
- Let Not Gate override connectors for 1-input gate
- Draw And Gate from Left Frame Button



New Component Base class called Comp

Comp_Lib/component.py

```
import tkinter as tk
from PIL import Image, ImageTk

from Helper_Lib import Point
from Wire_Lib.connector import Connector


class Comp:
    def __init__(self, canvas, x1, y1):
        """Base class for gate classes"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0
```

```

        self.is_selected = True
        self.is_drawing = True

        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []

    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is
selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

```

```

from pathlib import Path

from Comp_Lib.component import Comp
from Wire_Lib import Connector
from Helper_Lib import Point


class NotGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)

        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

    # Create 2 connectors
    self.in1_id, self.out1_id = None, None
    self.create_connectors()

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is
        selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
        self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
                                                    outline="red", width=2)

    def create_connectors(self):
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.out1_id = Connector(self.canvas, "out1", center.x + w / 2,
                                center.y) # Out1
        self.in1_id = Connector(self.canvas, "in1", center.x - w / 2, center.y)
# In1
        self.conn_list = [self.out1_id, self.in1_id]

```

```
from pathlib import Path

from Comp_Lib.component import Comp


class AndGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)

        self.filename = Path(__file__).parent / "../images/gates/and_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
```

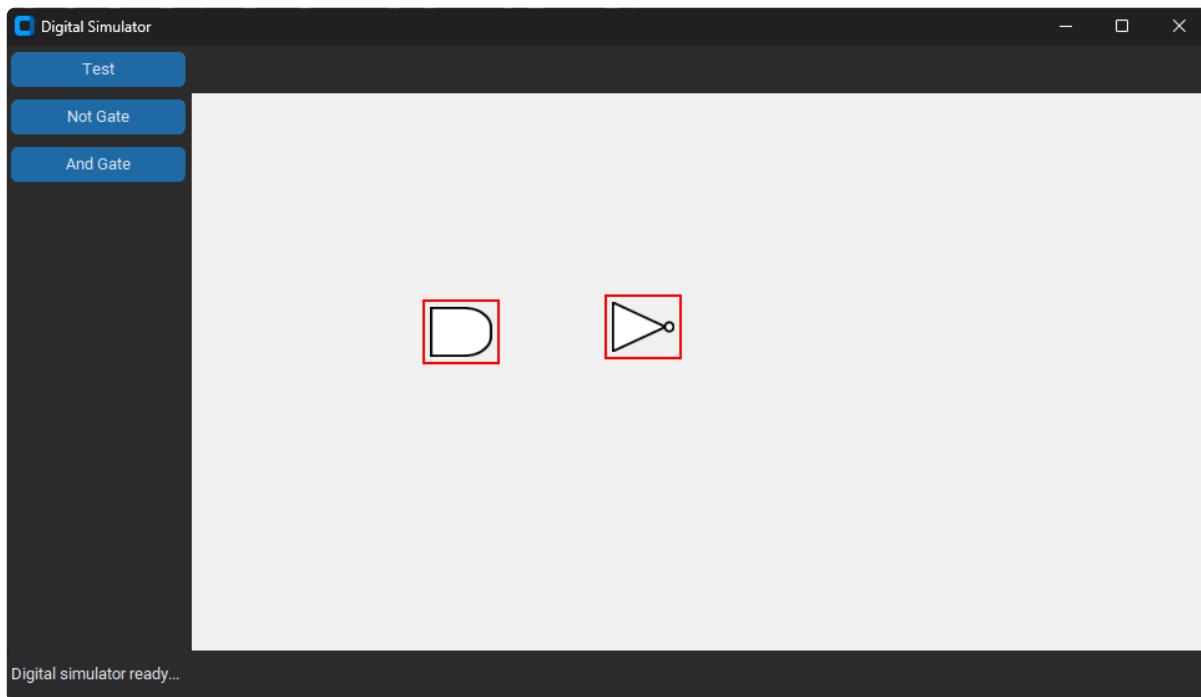
Notes:

- And gate class is very simple (for KISS)
- Base class has the methods used to create the gate
- The class basically sets the gate image file and calls common methods to create the gate

Move Gates with Mouse

Objectives:

- Select a component on the canvas
- Move the component with the left mouse button
- Update the component based on the mouse position



New Canvas Class

UI_Lib/canvas.py

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse


class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

        self.comp_list = []
        self.mouse = Mouse(self)
        self.mouse.move_mouse_bind_events()

    def redraw(self):
        for c in self.comp_list:
            c.update()
```

New Mouse Class

UI_Lib/mouse.py

```
from Helper_Lib import Point

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_comp = None

        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_mouse_bind_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)

        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                x2, y2 = self.selected_comp.x2, self.selected_comp.y2
                self.offset1.set(x - x1, y - y1)
                self.offset2.set(x - x2, y - y2)
            else:
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                # self.offset1.x, self.offset1.y =
        self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y)
        self.offset1.set(x - x1, y - y1)

    def move_left_drag(self, event):
        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1 = event.x - self.offset1.x
                y1 = event.y - self.offset1.y
                # x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
                x2 = event.x - self.offset2.x
                y2 = event.y - self.offset2.y
```

```

        # x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.selected_comp.x1, self.selected_comp.y1 = x1, y1
        self.selected_comp.x2, self.selected_comp.y2 = x2, y2
        self.canvas.redraw()

    else:
        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        # x, y = self.canvas.grid.snap_to_grid(x, y)
        self.selected_comp.x1, self.selected_comp.y1 = x, y
        self.canvas.redraw()

def move_left_up(self, _event):
    if self.selected_comp:
        self.offset1.set(0, 0)
        self.offset2.set(0, 0)

def select_hit_test(self, x, y):
    for s in self.canvas.comp_list:
        if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
            # print("Shape hit: ", s)
            self.selected_comp = s
            s.is_selected = True
            self.canvas.redraw()
            return

    # No shape hit - unselect all
    self.selected_comp = None
    self.unselect_all()

def unselect_all(self):
    for s in self.canvas.comp_list:
        s.is_selected = False
    self.canvas.redraw()

```

Add updates to Not Gate and And Gate Classes

Comp_Lib/component.py

```

import tkinter as tk
from PIL import Image, ImageTk

class Comp:

```

```

def __init__(self, canvas, x1, y1):
    """Base class for gate classes"""
    self.canvas = canvas
    self.x1 = x1
    self.y1 = y1

    self.id = None
    self.sel_id = None
    self.a_image = None
    self.ph_image = None
    self.bbox = None
    self.angle = 0
    self.filename = None

    self.is_selected = False
    self.is_drawing = False

    self.in1, self.in2, self.out = None, None, None
    self.conn_list = []

def create_image(self, filename):
    """Initial component image creation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True)
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

def update_position(self):
    """Update the position when the gate object is moved"""
    self.canvas.coords(self.id, self.x1, self.y1) # Update position

def update_image(self, filename):
    """Update the image for gate symbol rotation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True) # Update
image rotation
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.canvas.itemconfig(self.id, image=self.ph_image) # Update image

def update_bbox(self):
    """Update the bounding box to get current gate coordinates"""
    self.bbox = self.canvas.bbox(self.id)

def create_selector(self):
    """Create the red rectangle selector and check to see if the gate is

```

```

selected"""

    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
        self.set_selector_visibility()

def update_selector(self):
    """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.canvas.coords(self.sel_id, x1, y1, x2, y2)
        self.set_selector_visibility()

def set_selector_visibility(self):
    """Set the selector visibility state"""
    if self.is_selected:
        self.canvas.itemconfig(self.sel_id, state='normal')
    else:
        self.canvas.itemconfig(self.sel_id, state='hidden')

def set_connector_visibility(self):
    """Set the connector visibility state"""
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

```

Comp_Lib/not_gate.py

```

from pathlib import Path

from Comp_Lib.component import Comp
from Wire_Lib import Connector
from Helper_Lib import Point


class NotGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)

```

```

        self.filename = Path(__file__).parent / "../images/gates/not_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 2 connectors
        self.in1_id, self.out1_id = None, None
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.out1_id = Connector(self.canvas, "out1", center.x + w / 2,
center.y) # Out1
        self.in1_id = Connector(self.canvas, "in1", center.x - w / 2, center.y)
# In1
        self.conn_list = [self.out1_id, self.in1_id]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0 or self.angle == 180:
            sign_x = lambda x: -1 if (self.angle == 180) else 1 # Lambda
function to set sign of x
            self.out1_id.set_pos(center.x + sign_x(self.angle) * w / 2,
center.y)
            self.in1_id.set_pos(center.x - sign_x(self.angle) * w / 2,
center.y)

```

```

        elif self.angle == 90 or self.angle == 270:
            sign_y = lambda y: -1 if (self.angle == 270) else 1 # Lambda
    function to set sign of y
            self.out1_id.set_pos(center.x, center.y - sign_y(self.angle) * h /
2)
            self.in1_id.set_pos(center.x, center.y + sign_y(self.angle) * h /
2)

        for c in self.conn_list:
            c.update()

# self.move_connected_wires()

```

Comp_Lib/and_gate.py

```

from pathlib import Path
from Comp_Lib.component import Comp
from Helper_Lib import Point
from Wire_Lib.connector import Connector

class AndGate(Comp):
    """And Gate Model"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)

        self.filename = Path(__file__).parent / "../images/gates/and_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size

```

```

        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
            self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h/2
            self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
            self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
            self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2
            self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
            self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

        for c in self.conn_list:
            c.update()

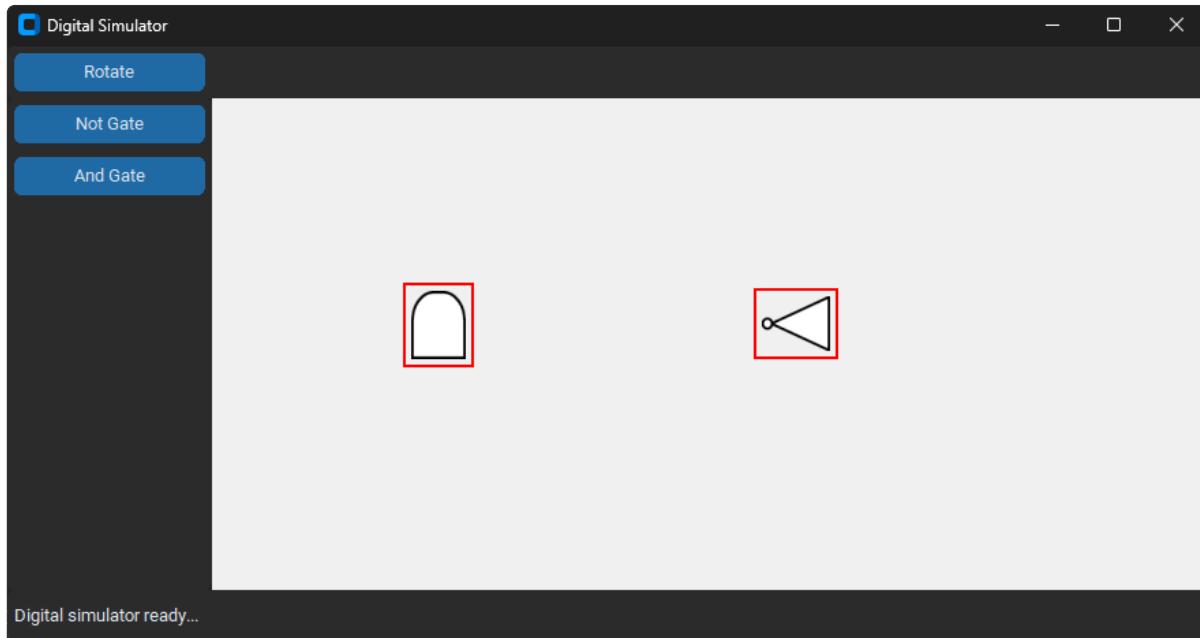
        # self.move_connected_wires()

```

Component Rotation

Objectives:

- Rotate selected shape from top frame button
- Rotate selected shape using 'r' key on the keyboard
- Rotate shape in 90 deg increments



Comp_Lib/component.py

```
...  
  
    def set_connector_visibility(self):  
        """Set the connector visibility state"""  
        if self.is_drawing:  
            for c in self.conn_list:  
                self.canvas.itemconfig(c.id, state='normal')  
        else:  
            for c in self.conn_list:  
                self.canvas.itemconfig(c.id, state='hidden')  
  
    def rotate(self): # Added new method  
        """Set the rotation angle to the current angle + 90 deg, reset to 0 deg  
        if angle > 270 deg"""  
        self.angle += 90  
        if self.angle > 270:  
            self.angle = 0
```

digital_simulator.py

```
    . . .

        # Add widgets to frames to cause the frames to auto-size
        top_frame_button = ctk.CTkButton(self.top_frame, text="Rotate",
    command=self.rotate_comp) # Modified to call the component rotation method
        top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

    . . .

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add bindings here
        self.bind('<r>', self.rotate_comp) # Added binding to the 'r' key

    def create_not_gate(self):
        self.gate1 = NotGate(self.canvas, 100, 100)
        self.canvas.comp_list.append(self.gate1)

    . . .

    def rotate_comp(self, event=None): # Added new component rotation method
        if self.canvas.mouse.selected_comp:
            self.canvas.mouse.selected_comp.rotate()
            self.canvas.redraw()

if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

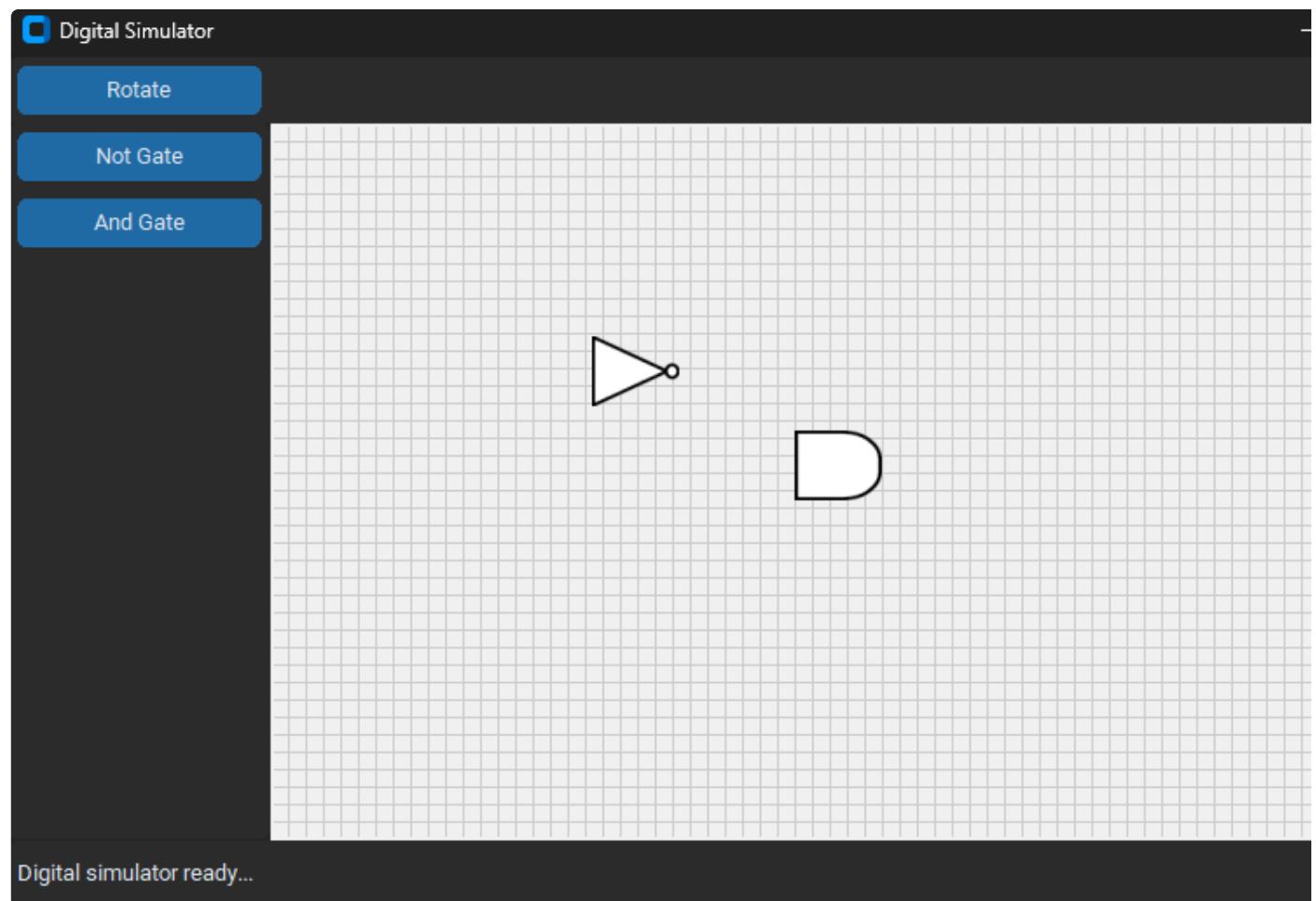
Note:

- Setting `event=None` in the `rotate_comp()` method allows it to be called as a button handler and a keyboard binding

Grid Class

Objectives:

- Draw a background grid on the canvas
- Add a snap-to-grid capability for all components



New Grid Class

UI_Lib/grid.py

```
class Grid:  
    def __init__(self, canvas, grid_size):  
        self.canvas = canvas  
        self.grid_size = grid_size  
        self.grid_visible = True  
  
        self.grid_snap = self.grid_size  
        self.draw()  
  
    def draw(self):
```

```

    if self.grid_visible:
        w = self.canvas.winfo_width() # Get current width of canvas
        h = self.canvas.winfo_height() # Get current height of canvas

        # Creates all vertical lines at intervals of 100
        for i in range(0, w, self.grid_size):
            self.canvas.create_line([(i, 0), (i, h)], fill="#cccccc",
tags='grid_line')

        # Creates all horizontal lines at intervals of 100
        for i in range(0, h, self.grid_size):
            self.canvas.create_line([(0, i), (w, i)], fill="#cccccc",
tags='grid_line')

    def snap_to_grid(self, x, y):
        if self.grid_visible:
            x = round(x / self.grid_snap) * self.grid_snap
            y = round(y / self.grid_snap) * self.grid_snap
        return x, y

```

UI_Lib/canvas.py

```

import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid # Added import for Grid class

class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

        self.comp_list = []
        self.mouse = Mouse(self)
        self.mouse.move_mouse_bind_events()

        self.grid = Grid(self, 10) # Added grid object instantiation

    def redraw(self):
        self.delete('grid_line') # Delete grid lines on each redraw
        self.grid.draw() # Draw the grid
        self.tag_lower("grid_line") # Lower the grid so it is in the
background
        for c in self.comp_list:

```

```
c.update()
```

UI_Lib/mouse.py

```
from Helper_Lib import Point

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_comp = None

        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

    def unbind_mouse_events(self):
        self.canvas.unbind("<Button-1>")
        self.canvas.unbind("<B1-Motion>")
        self.canvas.unbind("<ButtonRelease-1>")

    def move_mouse_bind_events(self):
        self.unbind_mouse_events()
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)

        if self.selected_comp:
            if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                x2, y2 = self.selected_comp.x2, self.selected_comp.y2
                self.offset1.set(x - x1, y - y1)
                self.offset2.set(x - x2, y - y2)
            else:
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
                self.offset1.x, self.offset1.y =
        self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y) # Added snap-to-
        grid call
            self.offset1.set(x - x1, y - y1)
```

```

def move_left_drag(self, event):
    if self.selected_comp:
        if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
            x1 = event.x - self.offset1.x
            y1 = event.y - self.offset1.y
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1) # Added snap-
to-grid call
            x2 = event.x - self.offset2.x
            y2 = event.y - self.offset2.y
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2) # Added snap-
to-grid call
            self.selected_comp.x1, self.selected_comp.y1 = x1, y1
            self.selected_comp.x2, self.selected_comp.y2 = x2, y2
            self.canvas.redraw()
    else:
        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        x, y = self.canvas.grid.snap_to_grid(x, y) # Added snap-to-
grid call
        self.selected_comp.x1, self.selected_comp.y1 = x, y
        self.canvas.redraw()

def move_left_up(self, _event):
    if self.selected_comp:
        self.offset1.set(0, 0)
        self.offset2.set(0, 0)

def select_hit_test(self, x, y):
    for s in self.canvas.comp_list:
        if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
            # print("Shape hit: ", s)
            self.selected_comp = s
            s.is_selected = True
            self.canvas.redraw()
            return

    # No shape hit - unselect all
    self.selected_comp = None
    self.unselect_all()

def unselect_all(self):
    for s in self.canvas.comp_list:
        s.is_selected = False
    self.canvas.redraw()

```

`digital_simulator.py`

```
    . . .

    # Add bindings here
    self.bind('<r>', self.rotate_comp)
    self.bind("<Configure>", self.on_window_resize) # Add binding to
window resize event

    . . .

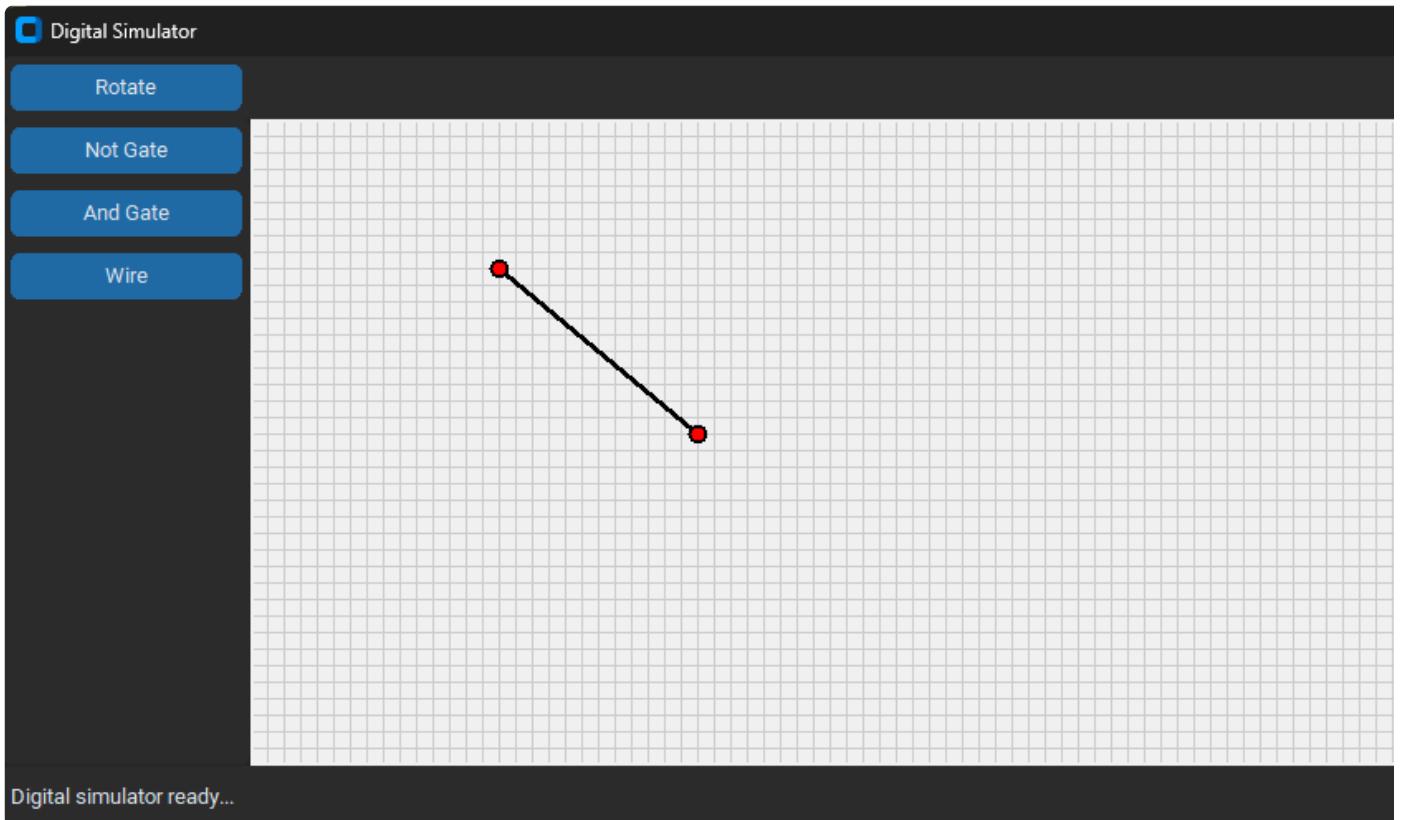
def on_window_resize(self, _event): # Add method to redraw canvas whenever
window is resized
    self.canvas.redraw()

if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()
```

Draw Wires with Mouse

Objectives:

- Create wires using the mouse to "draw" the wire while dragging the mouse
- Create the wire in a "wire group" regardless if there is one wire or more than one wire in the group
- Add wire to the wire group if it intersects with an existing wire
- Show gate connectors while drawing wires
- Connect wires to gate connectors and resize connected wires if the gate is moved or rotated
- Wire selectors will be red ovals located at the two ends of the wire



New Wire Class

```
from Wire_Lib.wire_selector import WireSelector

class Wire:
    def __init__(self, canvas, x1, y1, x2, y2):
        """Wire base class"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        # Wire appearance variables
        self.fill_color = "black"
        self.border_width = 3

        self.id = None
        self.state = False

        self.bbox = None
        self.is_selected = True
        self.sel_list = []
```

```

        self.selector = None

        self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
width=self.border_width, tags='wire')

        self.s1_id, self.s2_id = None, None
        self.create_selectors()
        self.set_selector_visibility()

def create_selectors(self):
    """Create selectors at the ends of the wire here"""
    self.s1_id = WireSelector(self.canvas, "begin", self.x1, self.y1)
    self.s2_id = WireSelector(self.canvas, "end", self.x2, self.y2)

    self.sel_list = [self.s1_id, self.s2_id]

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_selectors()

def update_position(self):
    """Update the position when the gate object is moved"""
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2) #
Update position

def update_bbox(self):
    self.bbox = self.canvas.bbox(self.id)

def update_border_width(self):
    self.canvas.itemconfig(self.id, width=self.border_width)

def update_selectors(self):
    """Update the position of all selectors here"""
    self.s1_id.x, self.s1_id.y = self.x1, self.y1
    self.s1_id.update()

    self.s2_id.x, self.s2_id.y = self.x2, self.y2
    self.s2_id.update()

def set_selector_visibility(self):
    if self.is_selected:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:

```

```

        self.canvas.itemconfig(s.id, state='hidden')

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
            # self.x2, self.y2 = self.canvas.grid.snap_to_grid(self.x2,
self.y2)
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
            # self.x1, self.y1 = self.canvas.grid.snap_to_grid(self.x1,
self.y1)

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Wire: " + " x1: " + str(self.x1) + " y1: " + str(self.y1) + \
               " x2: " + str(self.x2) + " y2: " + str(self.y2)

```

UI_Lib/mouse.py

```

from Helper_Lib import Point
from Wire_Lib import Connection # Added import for connection class

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas
        self.selected_comp = None
        self.current_wire_obj = None

        self.start = Point(0, 0)
        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

```

```

def unbind_mouse_events(self):
    self.canvas.unbind("<Button-1>")
    self.canvas.unbind("<B1-Motion>")
    self.canvas.unbind("<ButtonRelease-1>")

def move_mouse_bind_events(self):
    self.unbind_mouse_events()
    self.canvas.bind("<Button-1>", self.move_left_down)
    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

def draw_wire_mouse_events(self): # Added method to bind draw wire methods
    self.unbind_mouse_events()
    self.canvas.bind("<Button-1>", self.draw_left_down)
    self.canvas.bind("<B1-Motion>", self.draw_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

def move_left_down(self, event):
    x, y = event.x, event.y
    self.select_hit_test(x, y)

    if self.selected_comp:
        if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
            x1, y1 = self.selected_comp.x1, self.selected_comp.y1
            x2, y2 = self.selected_comp.x2, self.selected_comp.y2
            self.offset1.set(x - x1, y - y1)
            self.offset2.set(x - x2, y - y2)
        else:
            x1, y1 = self.selected_comp.x1, self.selected_comp.y1
            self.offset1.x, self.offset1.y =
    self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y)
    self.offset1.set(x - x1, y - y1)

def move_left_drag(self, event):
    if self.selected_comp:
        if self.canvas.gettags(self.selected_comp.id)[0] == 'wire':
            x1 = event.x - self.offset1.x
            y1 = event.y - self.offset1.y
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
            x2 = event.x - self.offset2.x
            y2 = event.y - self.offset2.y
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
            self.selected_comp.x1, self.selected_comp.y1 = x1, y1
            self.selected_comp.x2, self.selected_comp.y2 = x2, y2
            self.canvas.redraw()
    else:

```

```

        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.selected_comp.x1, self.selected_comp.y1 = x, y
        self.canvas.redraw()

def move_left_up(self, _event):
    if self.selected_comp:
        self.offset1.set(0, 0)
        self.offset2.set(0, 0)

def draw_left_down(self, event): # Added method for draw left down
    if self.current_wire_obj:
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y
        self.start.x, self.start.y =
self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

        self.current_wire_obj.x1, self.current_wire_obj.y1 = self.start.x,
self.start.y
        self.current_wire_obj.x2, self.current_wire_obj.y2 = self.start.x,
self.start.y

        if self.current_wire_obj is not None:
            self.select_connector(self.current_wire_obj, "begin",
self.start.x, self.start.y)

def draw_left_drag(self, event): # Added method for draw left drag
    if self.current_wire_obj:
        shape = self.current_wire_obj
        x, y = event.x, event.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        shape.x1, shape.y1 = self.start.x, self.start.y
        shape.x2, shape.y2 = x, y
        self.canvas.redraw()

def draw_left_up(self, event): # Added method for draw left up
    self.select_connector(self.current_wire_obj, "end", event.x, event.y)
    self.canvas.hide_connectors()
# self.move_mouse_bind_events()

def select_hit_test(self, x, y):
    for s in self.canvas.comp_list:
        if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
            # print("Shape hit: ", s)

```

```

        self.selected_comp = s
        s.is_selected = True
        self.canvas.redraw()
        return

    # No shape hit - unselect all
    self.selected_comp = None
    self.unselect_all()

def unselect_all(self):
    for s in self.canvas.comp_list:
        s.is_selected = False
    self.canvas.redraw()

def select_connector(self, wire_obj, wire_end, x, y): # Added method to
see if line end hits a gate connector
    for comp in self.canvas.comp_list:
        if not self.canvas.gettags(comp.id)[0] == 'wire':
            conn = comp.check_connector_hit(x, y)
            if conn:
                if wire_end == "begin":
                    wire_obj.x1, wire_obj.y1 = conn.x, conn.y
                elif wire_end == "end":
                    wire_obj.x2, wire_obj.y2 = conn.x, conn.y
                a_conn = Connection(conn, self.current_wire_obj, wire_end)
                comp.wire_list.append(a_conn)
                self.canvas.redraw()

```

`digital_simulator.py`

```

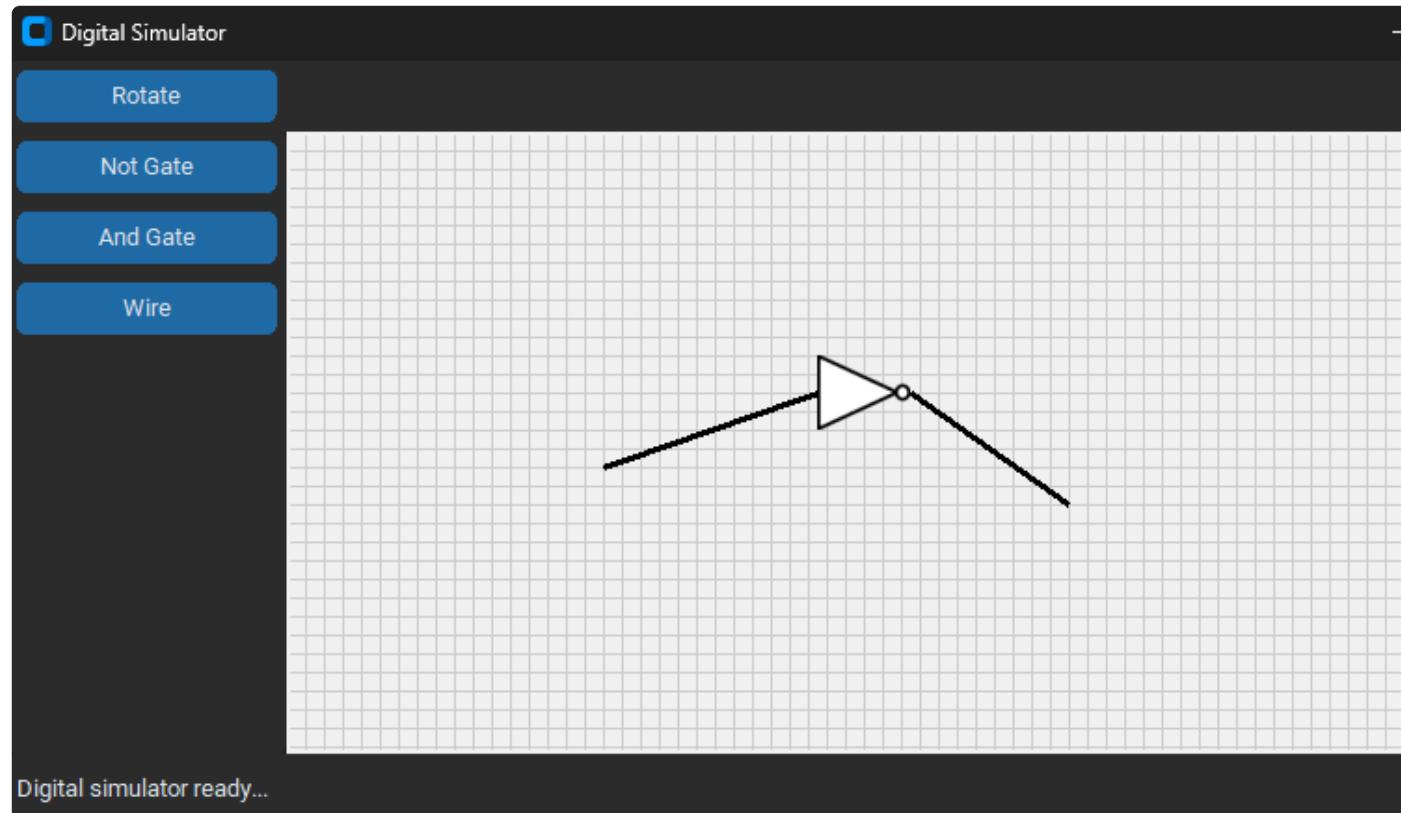
. . .

def create_wire(self): # updated method
    self.wire = Wire(self.canvas, 0, 0, 0, 0)
    self.canvas.comp_list.append(self.wire)
    self.canvas.mouse.current_wire_obj = self.wire
    self.canvas.mouse.draw_wire_mouse_events()

. . .

```

Move Wire if connected Gate moves



Comp_Lib/not_gate.py

```
    . . .
def update_connectors(self):
    . . .
    for c in self.conn_list:
        c.update()

    self.move_connected_wires() # Added call to move connected wires
```

Add the same call at end of the `update_connectors()` method in the And Gate Class

Comp_Lib/component.py

```

    . . .

def check_connector_hit(self, x, y): # Added new method
    """Hit test to see if a connector is at the provided x, y
coordinates"""
    for conn in self.conn_list:
        if conn.connector_hit_test(x, y):
            return conn
    return None

def move_connected_wires(self): # Added new method
    """Resize connected wires if the shape is moved"""
    for connection in self.wire_list:
        for connector in self.conn_list:
            if connector == connection.connector_obj:
                # print(connector, connection.line_obj, "Match")
                if connection.wire_end == "begin":
                    connection.wire_obj.x1 = connector.x
                    connection.wire_obj.y1 = connector.y
                elif connection.wire_end == "end":
                    connection.wire_obj.x2 = connector.x
                    connection.wire_obj.y2 = connector.y

```

Other Gates, Switch, LED, and Text Classes

Objectives:

- Create Nand Gate class
- Create Nor Gate class
- Create Xor Gate class
- Create Xnor Gate class
- Create Switch class
- Create LED class
 - Make LED color configurable
 - Make LED size configurable
- Create Text class

[UI_Lib/nand_gate.py](#)

```

from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class NandGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent /
"../images/gates/nand_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

    # Create connectors
    self.in1, self.in2, self.out = None, None, None
    self.conn_list = []
    self.create_connectors()
    self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +

```

h/4)

```
# Update the connector list
self.conn_list = [self.out, self.in1, self.in2]

def update_connectors(self):
    """Update the position of all connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    if self.angle == 0:
        self.out.x, self.out.y = center.x + w / 2, center.y
        self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
        self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
    elif self.angle == 90:
        self.out.x, self.out.y = center.x, center.y - h/2
        self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
        self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
    elif self.angle == 180:
        self.out.x, self.out.y = center.x - w / 2, center.y
        self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
        self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
    elif self.angle == 270:
        self.out.x, self.out.y = center.x, center.y + h / 2
        self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
        self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def set_logic_level(self):
    for wire in self.wire_list:
        if wire.connector_obj.name == "in1":
            self.in1_state = wire.wire_obj.state
            self.out_state = not(self.in1_state and self.in2_state)
        elif wire.connector_obj.name == "in2":
            self.in2_state = wire.wire_obj.state
            self.out_state = not(self.in1_state and self.in2_state)
        elif wire.connector_obj.name == "out":
            wire.wire_obj.state = self.out_state
```

UI_Lib/nor_gate.py

```
from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class NorGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/gates/nor_50x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

        # Create connectors
        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y -
```

```

h/4)
    self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y +
h/4)

    # Update the connector list
    self.conn_list = [self.out, self.in1, self.in2]

def update_connectors(self):
    """Update the position of all connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    if self.angle == 0:
        self.out.x, self.out.y = center.x + w / 2, center.y
        self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
        self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
    elif self.angle == 90:
        self.out.x, self.out.y = center.x, center.y - h/2
        self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
        self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
    elif self.angle == 180:
        self.out.x, self.out.y = center.x - w / 2, center.y
        self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
        self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
    elif self.angle == 270:
        self.out.x, self.out.y = center.x, center.y + h / 2
        self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
        self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def set_logic_level(self):
    for wire in self.wire_list:
        if wire.connector_obj.name == "in1":
            self.in1_state = wire.wire_obj.state
            self.out_state = not(self.in1_state or self.in2_state)
        elif wire.connector_obj.name == "in2":
            self.in2_state = wire.wire_obj.state
            self.out_state = not(self.in1_state or self.in2_state)
        elif wire.connector_obj.name == "out":

```

```
        wire.wire_obj.state = self.out_state
```

UI_Lib/xor_gate.py

```
from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class XorGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/gates/xor_60x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

        # Create connectors
        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
```

```

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y - h/4)
        self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y + h/4)

        # Update the connector list
        self.conn_list = [self.out, self.in1, self.in2]

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
            self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h/2
            self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
            self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
            self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
            self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2
            self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
            self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        for wire in self.wire_list:
            if wire.connector_obj.name == "in1":
                self.in1_state = wire.wire_obj.state
                self.out_state = self.in1_state != self.in2_state
            elif wire.connector_obj.name == "in2":

```

```

        self.in2_state = wire.wire_obj.state
        self.out_state = self.in1_state != self.in2_state
    elif wire.connector_obj.name == "out":
        wire.wire_obj.state = self.out_state

```

UI_Lib/xnor_gate.py

```

from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector


class XnorGate(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent /
"../images/gates/xnor_60x40.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        self.in1_state = False
        self.in2_state = False
        self.out_state = False

    # Create connectors
    self.in1, self.in2, self.out = None, None, None
    self.conn_list = []
    self.create_connectors()
    self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):

```

```

# Calculate position of connectors from current shape position and size
x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
w, h = x2 - x1, y2 - y1
center = Point(x1 + w / 2, y1 + h / 2)

# Define 3 connectors: in1, in2, out
self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y - h/4)
self.in2 = Connector(self.canvas, "in2", center.x - w / 2, center.y + h/4)

# Update the connector list
self.conn_list = [self.out, self.in1, self.in2]

def update_connectors(self):
    """Update the position of all connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    if self.angle == 0:
        self.out.x, self.out.y = center.x + w / 2, center.y
        self.in1.x, self.in1.y = center.x - w / 2, center.y - h/4
        self.in2.x, self.in2.y = center.x - w / 2, center.y + h/4
    elif self.angle == 90:
        self.out.x, self.out.y = center.x, center.y - h/2
        self.in1.x, self.in1.y = center.x + w / 4, center.y + h/2
        self.in2.x, self.in2.y = center.x - w / 4, center.y + h/2
    elif self.angle == 180:
        self.out.x, self.out.y = center.x - w / 2, center.y
        self.in1.x, self.in1.y = center.x + w / 2, center.y - h/4
        self.in2.x, self.in2.y = center.x + w / 2, center.y + h/4
    elif self.angle == 270:
        self.out.x, self.out.y = center.x, center.y + h / 2
        self.in1.x, self.in1.y = center.x + w / 4, center.y - h / 2
        self.in2.x, self.in2.y = center.x - w / 4, center.y - h / 2

    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def set_logic_level(self):
    for wire in self.wire_list:

```

```

        if wire.connector_obj.name == "in1":
            self.in1_state = wire.wire_obj.state
            self.out_state = not(self.in1_state != self.in2_state)
        elif wire.connector_obj.name == "in2":
            self.in2_state = wire.wire_obj.state
            self.out_state = not(self.in1_state != self.in2_state)
        elif wire.connector_obj.name == "out":
            wire.wire_obj.state = self.out_state

```

UI_Lib/switch.py

```

from pathlib import Path
from PIL import Image

from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector
from Helper_Lib import Point


class Switch(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "switch"
        self.out_state = False # OFF state
        self.subckt_conn = None

        self.filename_sw_on = Path(__file__).parent /
"../images/switch/switch_on.png"
        self.on_image = Image.open(self.filename_sw_on)
        self.filename_sw_off = Path(__file__).parent /
"../images/switch/switch_off.png"
        self.off_image = Image.open(self.filename_sw_off)
        self.filename = self.filename_sw_off

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

    # Create 2 connectors
    self.out1_id = None
    self.create_connectors()
    self.set_connector_visibility()

```

```

def update(self):
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def toggle_switch(self):
    if self.filename == self.filename_sw_off:
        self.filename = self.filename_sw_on
    else:
        self.filename = self.filename_sw_off

def create_connectors(self): # Added new method
    """Create connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    # Define 1 connector: out1
    self.out1_id = Connector(self.canvas, "out", center.x + w / 2,
center.y)
    self.conn_list = [self.out1_id]
    self.set_connector_visibility()

def update_connectors(self): # Added new method
    """Update the position of all connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    # Update connector position based on angle
    if self.angle == 0:
        self.out1_id.x, self.out1_id.y = center.x + w / 2, center.y
    elif self.angle == 90:
        self.out1_id.x, self.out1_id.y = center.x, center.y - h / 2
    elif self.angle == 180:
        self.out1_id.x, self.out1_id.y = center.x - w / 2, center.y
    elif self.angle == 270:
        self.out1_id.x, self.out1_id.y = center.x, center.y + h / 2

    for c in self.conn_list:
        c.update()

```

```

        self.move_connected_wires()

    def toggle_state(self):
        self.out_state = not self.out_state
        self.toggle_switch()

        if self.wire_list:
            self.wire_list[0].wire_obj.state = self.out_state
            self.wire_list[0].wire_obj.set_wire_state()

```

UI_Lib/led.py

```

from pathlib import Path

from Wire_Lib.connector import Connector
from Comp_Lib.component import Comp
from Helper_Lib.point import Point


class LED(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "led"
        self.state = False
        self.angle = 180

        self.led_color = self.canvas.led_color # red, green, yellow, blue
        self.led_size = self.canvas.led_size # small, large

        if self.led_size == "large":
            self.w = 40
            self.h = 40
            color_led_str = "../images/led/led_on_" + self.led_color +
            "_large.png"
            self.filename_led_on = Path(__file__).parent / color_led_str
            self.filename_led_off = Path(__file__).parent /
            "../images/led/led_off_large.png"
        elif self.led_size == "small":
            self.w = 20
            self.h = 20
            color_led_str = "../images/led/led_on_" + self.led_color +
            "_small.png"

```

```

        self.filename_led_on = Path(__file__).parent / color_led_str
        self.filename_led_off = Path(__file__).parent /
"../images/led/led_off_small.png"
        self.filename = self.filename_led_off

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

# Create 2 connectors
self.in1_id = None
self.create_connectors()
self.set_connector_visibility()

def update(self):
    self.set_logic_level()
    self.update_state()
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def update_state(self):
    if self.state:
        self.filename = self.filename_led_on
    else:
        self.filename = self.filename_led_off

def create_connectors(self): # Added new method
    """Create connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

# Define 1 connector: out
self.in1_id = Connector(self.canvas, "in", center.x - w / 2, center.y)
self.conn_list = [self.in1_id]
self.set_connector_visibility()

def update_connectors(self): # Added new method
    """Update the position of all connectors here"""
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1

```

```

        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Update connector position based on angle
        if self.angle == 0:
            self.in1_id.x, self.in1_id.y = center.x + w / 2, center.y
        elif self.angle == 90:
            self.in1_id.x, self.in1_id.y = center.x, center.y - h / 2
        elif self.angle == 180:
            self.in1_id.x, self.in1_id.y = center.x - w / 2, center.y
        elif self.angle == 270:
            self.in1_id.x, self.in1_id.y = center.x, center.y + h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def set_logic_level(self):
        if self.wire_list:
            self.state = self.wire_list[0].wire_obj.state

```

UI_Lib/text.py

```

from Comp_Lib.component import Comp

class TextShape(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "text_shape"
        self.text = 'Hello World'

        self.fill_color = "black"
        self.border_color = "black"
        self.border_width = 3

        self.id = self.canvas.create_text(self.x1, self.y1,
                                         text=self.text, fill=self.fill_color,
                                         font='Helvetica 15 bold',
                                         angle=self.angle, tags="text")

    def update_bbox():

```

```

    self.create_selector()

def update(self):
    self.update_position()
    self.update_rotation()
    self.update_bbox()
    self.update_selector()

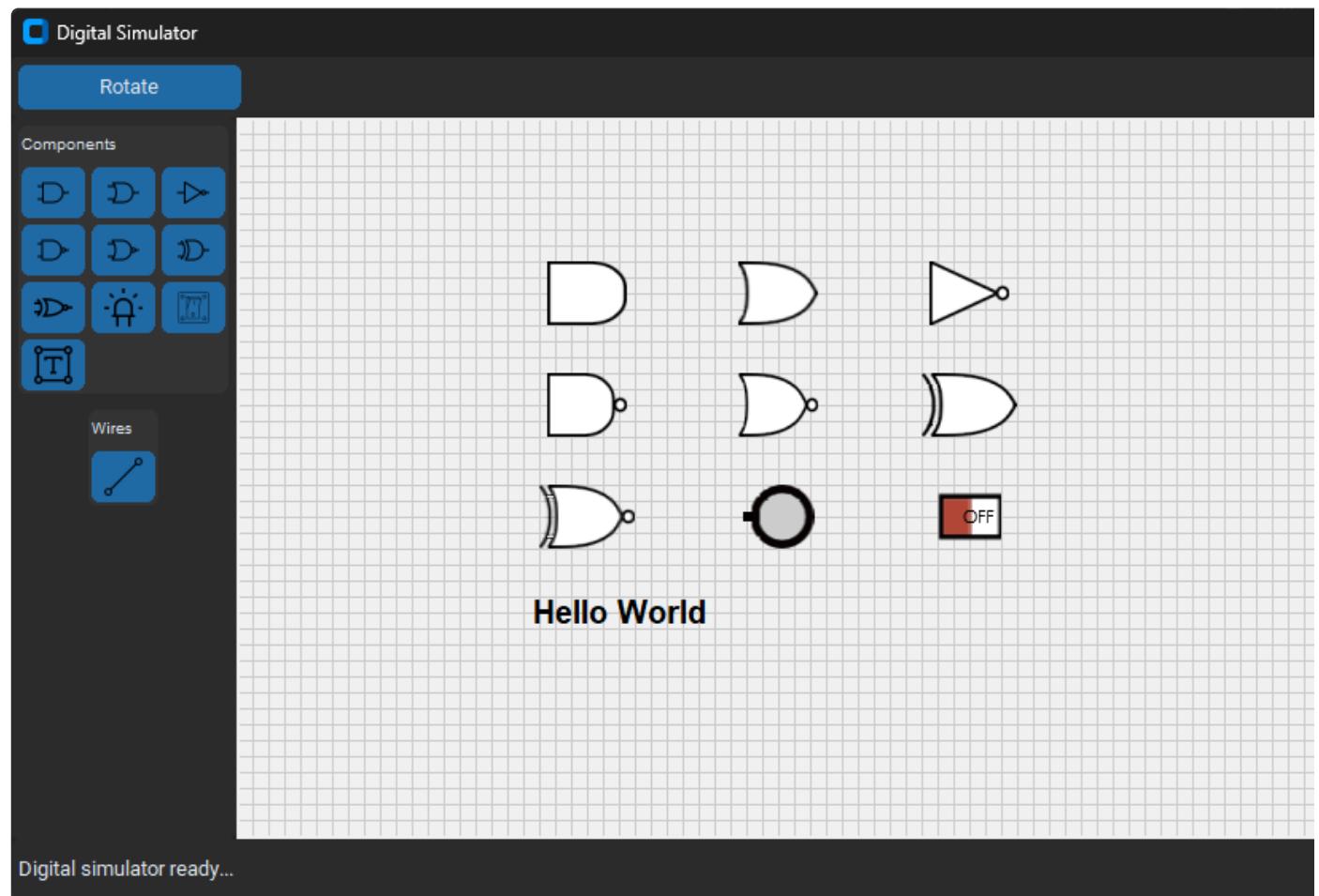
def update_rotation(self):
    self.canvas.itemconfig(self.id, angle=self.angle)

```

Left Frame Class

Objectives:

- Create a custom left frame class
- Component button frame with picture buttons for components in a grid
- Wire button frame with picture buttons for wires in a grid



digital_simulator.py

```
import customtkinter as ctk
from UI_Lib import Canvas, LeftFrame # Added import for left frame class

ctk.set_appearance_mode("System") # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue") # Themes: "blue" (standard), "green",
"dark-blue"

class DigitalSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Digital Simulator")

        self.canvas = Canvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = LeftFrame(self, self.canvas) # Modified to use the
left frame class
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add widgets to frames to cause the frames to auto-size
        top_frame_button = ctk.CTkButton(self.top_frame, text="Rotate",
command=self.rotate_comp) # Removed left frame buttons
        top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...")
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add bindings here
        self.bind('<r>', self.rotate_comp)
        self.bind("<Configure>", self.on_window_resize)

    def rotate_comp(self, _event=None):
        if self.canvas.mouse.selected_comp:
```

```

        self.canvas.mouse.selected_comp.rotate()
        self.canvas.redraw()

    def on_window_resize(self, _event):
        self.canvas.redraw()

if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()

```

UI_Lib/left_frame.py - new class

```

import customtkinter as ctk
from UI_Lib.comp_button_frame import CompButtonFrame
from UI_Lib.wire_button_frame import WireButtonFrame

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

        self.comp_button_frame = CompButtonFrame(self, self.canvas)
        self.comp_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.wire_button_frame = WireButtonFrame(self, self.canvas)
        self.wire_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

```

UI_Lib/comp_button_frame.py

```

import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from Comp_Lib import AndGate, OrGate, NotGate, NandGate, NorGate, XorGate,
XnorGate
from Comp_Lib import LED, Switch, Text

```

```

class CompButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None

        self.button_list = [("and_gate", "../icons/and.png"),
                            ("or_gate", "../icons/or.png"),
                            ("not_gate", "../icons/not.png"),
                            ("nand_gate", "../icons/nand.png"),
                            ("nor_gate", "../icons/nor.png"),
                            ("xor_gate", "../icons/xor.png"),
                            ("xnor_gate", "../icons/xnor.png"),
                            ("led", "../icons/led.png"),
                            ("switch", "../icons/switch.png"),
                            ("text", "../icons/text.png")]

    self.init_frame_widgets()

    def init_frame_widgets(self):
        frame_name_label = ctk.CTkLabel(self, text="Components", font=("Helvetica", 10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W, padx=2, pady=2)

        row_num, col_num = 1, 0
        for button in self.button_list:
            a_image = ctk.CTkImage(light_image=Image.open
                                    (Path(__file__).parent / button[1]),
                                    dark_image=Image.open
                                    (Path(__file__).parent / button[1]),
                                    size=(24, 24))
            self.button_id = ctk.CTkButton(self, text="", image=a_image,
                                           width=30,
                                           command=lambda
                                               a_name=button[0]:self.create_events(a_name))
            self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W, padx=2, pady=2)
            ToolTip(self.button_id, msg=button[0])
            row_num, col_num = self.update_grid_numbers(row_num, col_num)

    def create_events(self, name):
        gate = None

```

```

    if name == "and_gate":
        gate = AndGate(self.canvas, 100, 100)
    elif name == "or_gate":
        gate = OrGate(self.canvas, 100, 100)
    elif name == "not_gate":
        gate = NotGate(self.canvas, 100, 100)
    elif name == "nand_gate":
        gate = NandGate(self.canvas, 100, 100)
    elif name == "nor_gate":
        gate = NorGate(self.canvas, 100, 100)
    elif name == "xor_gate":
        gate = XorGate(self.canvas, 100, 100)
    elif name == "xnor_gate":
        gate = XnorGate(self.canvas, 100, 100)
    elif name == "led":
        gate = LED(self.canvas, 100, 100)
    elif name == "switch":
        gate = Switch(self.canvas, 100, 100)
    elif name == "text":
        gate = Text(self.canvas, 100, 100)
    self.canvas.comp_list.append(gate)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

@staticmethod
def update_grid_numbers(row, column):
    column += 1
    if column > 2:
        column = 0
        row += 1
    return row, column

```

UI_Lib/wire_button_frame.py

```

import customtkinter as ctk
from pathlib import Path
from PIL import Image

from Wire_Lib import Wire


class WireButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):

```

```

super().__init__(parent)
self.parent = parent
self.canvas = canvas
self.wire = None

# Add frame widgets here
frame_name_label = ctk.CTkLabel(self, text="Wires", font=("Helvetica",
10), height=20)
frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

wire_image = ctk.CTkImage(light_image=Image.open
(Path(__file__).parent /
"../icons/straight_line.png"),
dark_image=Image.open
(Path(__file__).parent /
"../icons/straight_line.png"),
size=(24, 24))

wire1_button = ctk.CTkButton(self, text="", image=wire_image, width=30,
command=self.create_wire)
wire1_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)

# Shape button handlers
def create_wire(self):
    wire = Wire(self.canvas, 0, 0, 0, 0)
    self.canvas.comp_list.append(wire)
    self.canvas.mouse.current_wire_obj = wire
    self.canvas.show_connectors()
    self.canvas.mouse.draw_wire_mouse_events()

```

UI.Lib/canvas.py

```

import customtkinter as ctk

from UI.Lib.mouse import Mouse
from UI.Lib.grid import Grid


class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

```

```

        self.led_color = "red"
        self.led_size = "large"
        self.comp_list = []
        self.mouse = Mouse(self)
        self.mouse.move_mouse_bind_events()

    self.grid = Grid(self, 10)

    def redraw(self):
        self.delete('grid_line')
        self.grid.draw()
        self.tag_lower("grid_line")
        for c in self.comp_list:
            c.update()

    def show_connectors(self):
        for s in self.comp_list:
            s.is_drawing = True
        self.redraw()

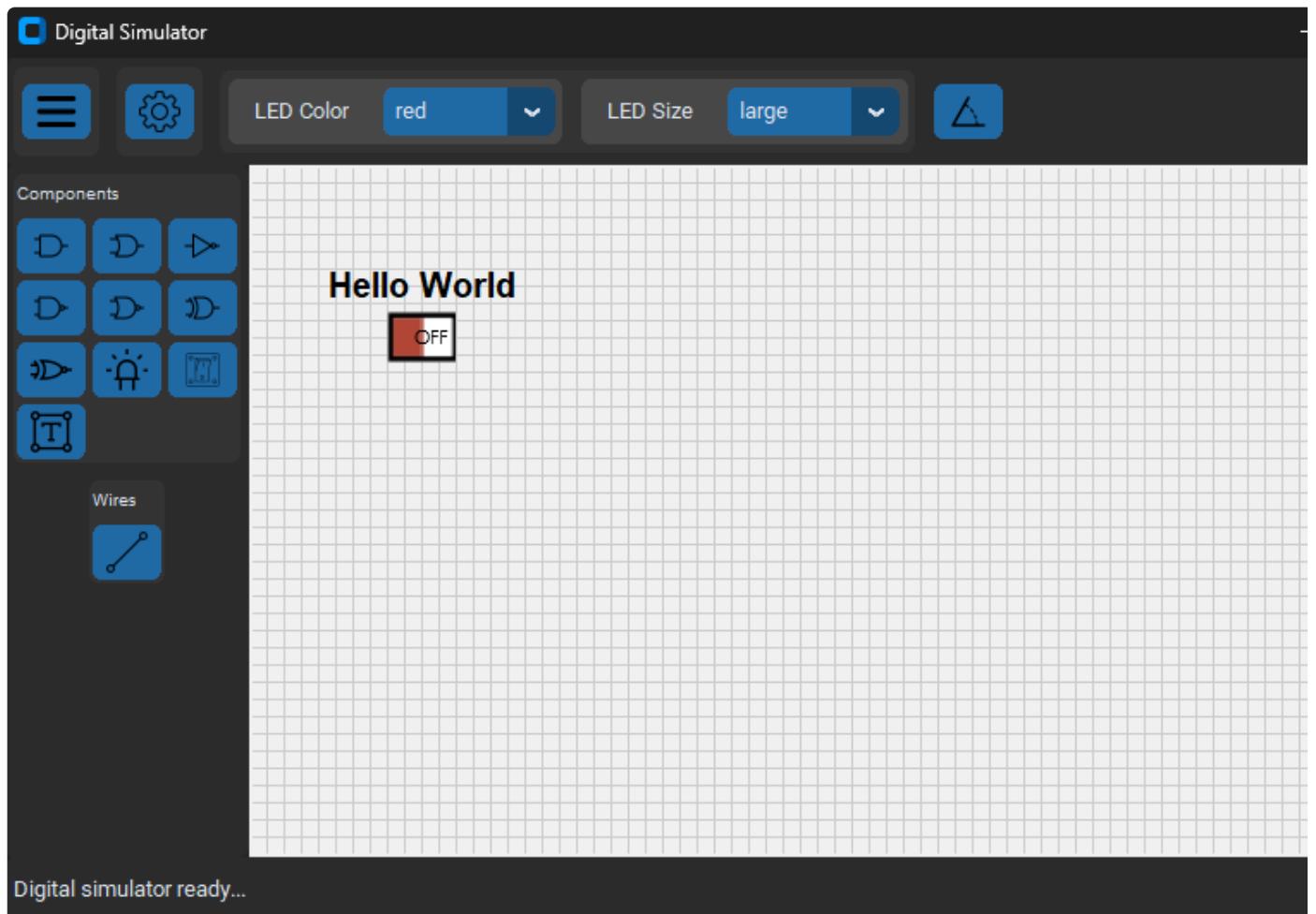
    def hide_connectors(self):
        for s in self.comp_list:
            s.is_drawing = False
        self.redraw()

```

Top Frame Class

Objectives:

- Create a custom Top Frame class
- Add file menu control
- Add settings control
- Add LED color and size control
- Add rotation button
- Add help menu control



digital_simulator.py

```
import customtkinter as ctk
from UI_Lib import Canvas, LeftFrame, TopFrame # added import for top frame
class

    ctk.set_appearance_mode("System") # Modes: "System" (standard), "Dark",
    "Light"
    ctk.set_default_color_theme("blue") # Themes: "blue" (standard), "green",
    "dark-blue"

class DigitalSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Digital Simulator")
```

```

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self, self.canvas) # Modified to use the top
frame class
        self.left_frame = LeftFrame(self, self.canvas)
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

# Add widgets to frames to cause the frames to auto-size
bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Digital
simulator ready...") # Removed top frame buttons
bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

# Add bindings here
self.bind('<r>', self.rotate_comp)
self.bind("<Configure>", self.on_window_resize)

def rotate_comp(self, _event=None):
    if self.canvas.mouse.selected_comp:
        self.canvas.mouse.selected_comp.rotate()
        self.canvas.redraw()

def on_window_resize(self, _event):
    self.canvas.redraw()

if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = DigitalSimulatorApp()
    app.mainloop()

```

UI_Lib/top_frame.py - new class

```

import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from UI_Lib.file_menu_frame import FileMenuFrame
from UI_Lib.settings_frame import SettingsFrame

```

```

from UI_Lib.led_frame import LedFrame
from UI_Lib.help_frame import HelpFrame


class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add Top Frame widget here
        file_frame = FileMenuFrame(self.parent, self, self.canvas)
        file_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        settings_frame = SettingsFrame(self.parent, self, self.canvas)
        settings_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        led_frame = LedFrame(self.parent, self, self.canvas)
        led_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        help_frame = HelpFrame(self.parent, self, self.canvas)
        help_frame.pack(side=ctk.RIGHT, padx=5, pady=5)

        a_image = ctk.CTkImage(light_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
                               dark_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
                               size=(24, 24))
        self.button_id = ctk.CTkButton(self, text="", image=a_image, width=30,
command=self.rotate_comp)
        self.button_id.pack(side=ctk.LEFT, padx=5, pady=5)
        ToolTip(self.button_id, msg="Rotate selected component")

    def rotate_comp(self):
        self.parent.rotate_comp(_event=None)

```

UI_Lib/file_menu_frame.py - new class

```

import customtkinter as ctk
from tkinter import filedialog as fd
from pathlib import Path
import json
from PIL import Image

```

```
from Comp_Lib import AndGate, OrGate, NandGate, NorGate, XorGate, XnorGate, Switch, LED, Text
from Wire_Lib import Wire


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.obj_type_dict = {'and': AndGate,
                             'nand': NandGate,
                             'or': OrGate,
                             'nor': NorGate,
                             'xor': XorGate,
                             'xnor': XnorGate,
                             'switch': Switch,
                             'wire': Wire,
                             'led': LED,
                             'text': Text}

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
                               command=self.new_diagram)
        new_btn.pack(pady=5)

        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
                               command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
                               command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
                               command=window.destroy)
        exit_btn.pack(pady=5)

        my_image = ctk.CTkImage(light_image=Image.open
                               (Path(__file__).parent /
                               "../icons/hamburger_menu.png"),
                               dark_image=Image.open
```

```

                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                     size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def new_diagram(self):
        self.canvas.delete("all")
        self.canvas.comp_list = []

    def load_diagram(self):
        try:
            filetypes = (('json files', '*.json'), ('All files', '*.*'))
            f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
            with open(f) as file:
                obj_dict = json.load(file)
                for obj_type, attributes in obj_dict.items():
                    if obj_type.split()[0] == "wire":
                        obj = self.obj_type_dict[obj_type.split()[0]]
                        (self.parent.parent.active_canvas,
                         attributes[0], attributes[1],
                         attributes[2], attributes[3])
                         else:
                            obj = self.obj_type_dict[obj_type.split()[0]]
                        (self.parent.parent.active_canvas,
                         attributes[0], attributes[1])
                            obj.angle = attributes[3]
                            self.parent.parent.active_canvas.comp_list.append(obj)
                            self.canvas.redraw()

        except FileNotFoundError:
            with open('untitled.canvas', 'w') as _file:
                pass
            self.canvas.comp_list = []

    def save_diagram(self):
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
        with open(f, 'w') as file:
            obj_dict = {f'{obj.type} {id}': (obj.x1, obj.y1, obj.x2, obj.y2,
obj.angle) for

```

```

                id, obj in
enumerate(self.parent.parent.active_canvas.comp_list)}
    json.dump(obj_dict, file)

def show_menu(self):
    if not self.menu_on:
        self.menu_frame.place(x=15, y=60)
        self.menu_frame.tkraise()
        self.menu_on = True
    else:
        self.menu_frame.place_forget()
        self.menu_on = False

```

UI_Lib/settings_frame.py - new class

```

import customtkinter as ctk
from PIL import Image

class SettingsFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        def grid_switch_event():
            if canvas.grid.grid_visible:
                self.canvas.grid.grid_visible = False
            else:
                self.canvas.grid.grid_visible = True
            self.canvas.redraw()

        switch_var = ctk.StringVar(value="on")
        switch = ctk.CTkSwitch(self.menu_frame, text="Grid",
command=grid_switch_event,
                                         variable=switch_var, onvalue="on",
offvalue="off")
        switch.pack(padx=5, pady=5)

```

```

        grid_size_label = ctk.CTkLabel(self.menu_frame, text="Grid Size", font=\
("Helvetica", 10), height=20)
        grid_size_label.pack(padx=5, pady=5, anchor="w")

    def optionmenu_callback(choice):
        self.canvas.grid.grid_size = int(choice)
        self.canvas.redraw()

    optionmenu = ctk.CTkOptionMenu(self.menu_frame, values=[ "5", "10",
"20", "30", "40", "50"], command=optionmenu_callback)
    optionmenu.pack(padx=5, pady=5)
    optionmenu.set("10")

    grid_snap_label = ctk.CTkLabel(self.menu_frame, text="Snap Size", font=\
("Helvetica", 10), height=20)
    grid_snap_label.pack(padx=5, pady=5, anchor="w")

    def snap_option_callback(choice):
        if choice == "Grid Size":
            self.canvas.grid.grid_snap = canvas.grid.grid_size
        else:
            self.canvas.grid.grid_snap = int(choice)
        canvas.redraw()

    snap_option = ctk.CTkOptionMenu(self.menu_frame, values=[ "Grid Size",
"5", "10", "20", "30", "40", "50"], command=snap_option_callback)
    snap_option.pack(padx=5, pady=5)
    snap_option.set("Grid Size")

    self.appearance_mode_label = ctk.CTkLabel(self.menu_frame,
text="Appearance Mode:", anchor="w")
    self.appearance_mode_label.pack(padx=5, pady=5)
    self.appearance_mode_optionemenu = ctk.CTkOptionMenu(self.menu_frame,
values=[ "Light", "Dark", "System"],

command=self.change_appearance_mode_event)
    self.appearance_mode_optionemenu.pack(padx=5, pady=5)
    self.appearance_mode_optionemenu.set("Dark")

    my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/DiagramEditor/icons/settings.png"),
dark_image=Image.open

```

```

("D:/EETools/DiagramEditor/icons/settings.png"),
    size=(24, 24))

        button = ctk.CTkButton(self, text="",
            image=my_image, width=30,
            command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

    @staticmethod
    def change_appearance_mode_event(new_appearance_mode: str):
        ctk.set_appearance_mode(new_appearance_mode)

```

UI_Lib/led_frame.py - new class

```

import customtkinter as ctk
from tktooltip import ToolTip

class LedFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        self.led_color = "red" # red, yellow, blue, green
        self.led_size = "large" # large, small

        self.init_led_color_control(self)
        self.init_led_size_control(self)

    def init_led_color_control(self, parent_frame):
        move_frame = ctk.CTkFrame(parent_frame, width=150)
        move_frame.configure(fg_color=("gray28", "gray28")) # set frame color
        move_frame.pack(side=ctk.LEFT, padx=5, pady=5)

```

```

        image_label = ctk.CTkLabel(move_frame, text="LED Color",
corner_radius=10)
        image_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add OptionMenu to top frame
        def option_menu_callback(choice):
            self.canvas.led_color = choice

            option_menu = ctk.CTkOptionMenu(move_frame, values=["red", "yellow",
"blue", "green"], width=100,
                                         command=option_menu_callback)
            option_menu.pack(side=ctk.LEFT, padx=5, pady=5)
            option_menu.set("red")

            ToolTip(option_menu, msg="Set LED color")

    def init_led_size_control(self, parent_frame):
        move_frame = ctk.CTkFrame(parent_frame, width=150)
        move_frame.configure(fg_color=("gray28", "gray28")) # set frame color
        move_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        image_label = ctk.CTkLabel(move_frame, text="LED Size",
corner_radius=10)
        image_label.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add OptionMenu to top frame
        def option_menu_callback(choice):
            self.canvas.led_size = choice

            option_menu = ctk.CTkOptionMenu(move_frame, values=["large", "small"],
width=100,
                                         command=option_menu_callback)
            option_menu.pack(side=ctk.LEFT, padx=5, pady=5)
            option_menu.set("large")

            ToolTip(option_menu, msg="Set LED size")

```

UI_Lib/help_frame.py - new class

```

import customtkinter as ctk
from tkinter import messagebox
from PIL import Image

```

```
class HelpFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.window = window
        self.parent = parent
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        about_image = ctk.CTkImage(light_image=Image.open

            ("D:/EETools/SimpleDiagramEditor/icons/about.png"),
            dark_image=Image.open

            ("D:/EETools/SimpleDiagramEditor/icons/about.png"),
            size=(24, 24))

        about_button = ctk.CTkButton(self.menu_frame, text="About",
image=about_image, width=30,
                                command=self.show_about_dialog)
        about_button.pack(side=ctk.TOP, padx=5, pady=5)

        my_image = ctk.CTkImage(light_image=Image.open

            ("D:/EETools/SimpleDiagramEditor/icons/help.png"),
            dark_image=Image.open

            ("D:/EETools/SimpleDiagramEditor/icons/help.png"),
            size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            menu_pos_x = self.canvas.winfo_width()
            self.menu_frame.place(x=menu_pos_x + 50, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
```

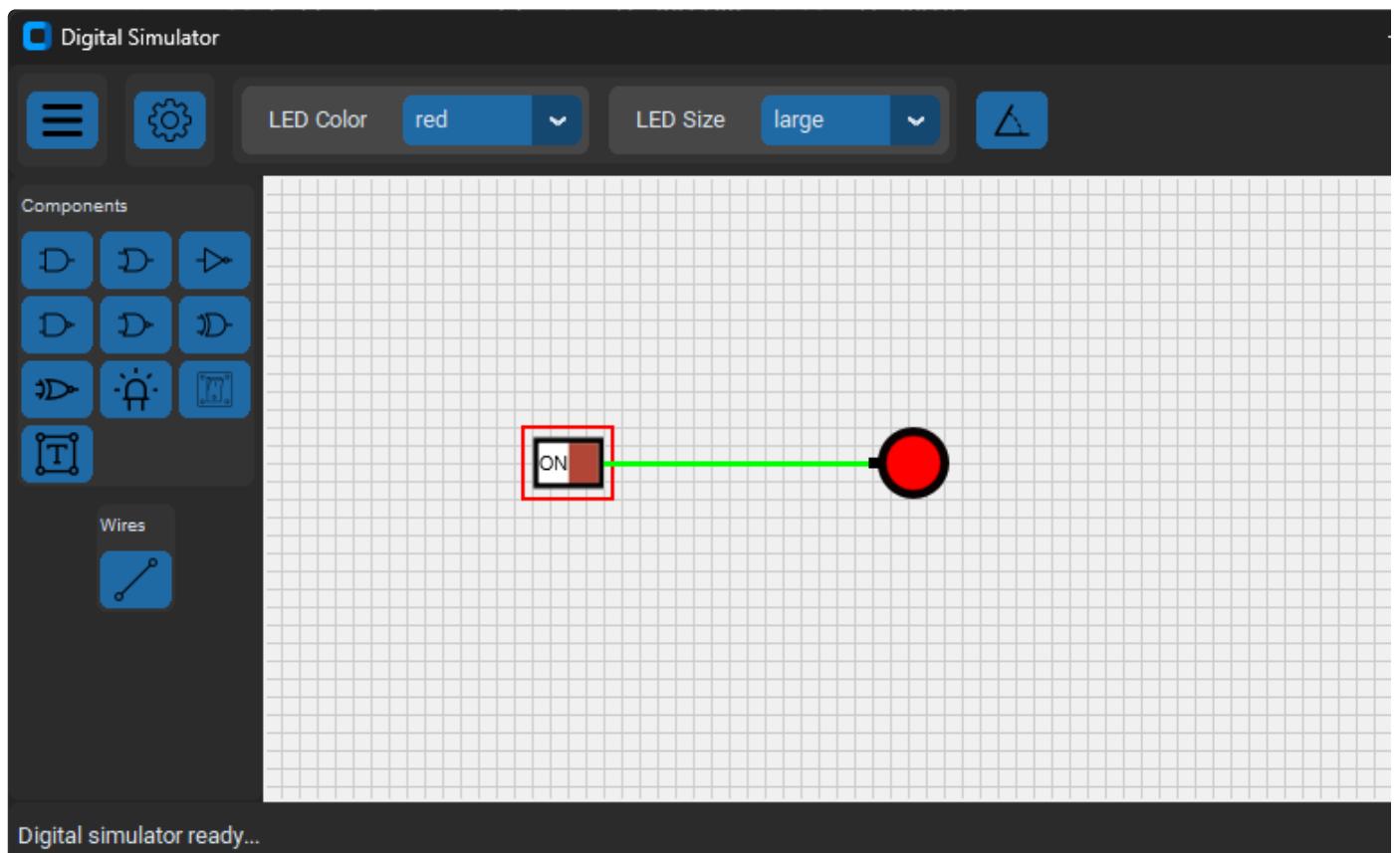
```
self.menu_on = False

@staticmethod
def show_about_dialog():
    messagebox.showinfo("About Digital Simulator", "Digital Simulator
v0.1\n" +
                        "Author: Rick A. Crist\n" + "2023")
```

Switch-LED Circuit Logic

Objectives:

- Create a circuit diagram with a switch, wire, and LED
- Toggle the switch with the right mouse button
- Propagate the logic signal from the switch to the wire to the LED



digital_simulator.py

```
# Add bindings here
self.bind('<r>', self.rotate_comp)
self.bind("<Configure>", self.on_window_resize)
self.canvas.bind('<Button-3>', self.canvas.edit_shape) # added new
binding to right mouse button

. . .
```

UI_Lib/canvas.py

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid
from Comp_Lib import Switch, Text # Added new import for switch and text
classes

. . .

def edit_shape(self, _event=None): # Added new method
    if isinstance(self.mouse.selected_comp, Switch):
        self.mouse.selected_comp.toggle_state()
        self.redraw()
    elif isinstance(self.mouse.selected_comp, Text):
        pass
```

Comp_Lib/switch.py

```
. . .

def toggle_state(self): # Method to toggle the output state
    self.out_state = not self.out_state
    self.toggle_switch()

    if self.wire_list:
        self.wire_list[0].wire_obj.state = self.out_state
```

Wire_Lib/wire.py

```

from Wire_Lib.wire_selector import WireSelector


class Wire:
    def __init__(self, canvas, x1, y1, x2, y2):
        """Wire base class"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        # Wire appearance variables
        self.fill_color = "black"
        self.border_width = 3

        self.id = None
        self.state = False # Variable to store the wire logic state

    . . .

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_wire_color() # Added call to update the wire color
        self.update_selectors()
        self.set_selector_visibility()

    . . .

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def update_wire_color(self): # Method to update wire color based on logic
        state
        if self.state:
            self.canvas.itemconfig(self.id, fill="#00ff00")
        else:
            self.canvas.itemconfig(self.id, fill=self.fill_color)

    . . .

```

```

. . .

class LED(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "led"
        self.state = False # Variable to store the led logic state
        self.angle = 180

. . .

    def update(self):
        self.set_logic_level() # Call set logic level method to update the led
logic state
        self.update_led_color() # Call to update state
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def update_led_color(self): # Method to update led color based on the led
state
        if self.state:
            self.filename = self.filename_led_on
        else:
            self.filename = self.filename_led_off

. . .

        self.move_connected_wires()

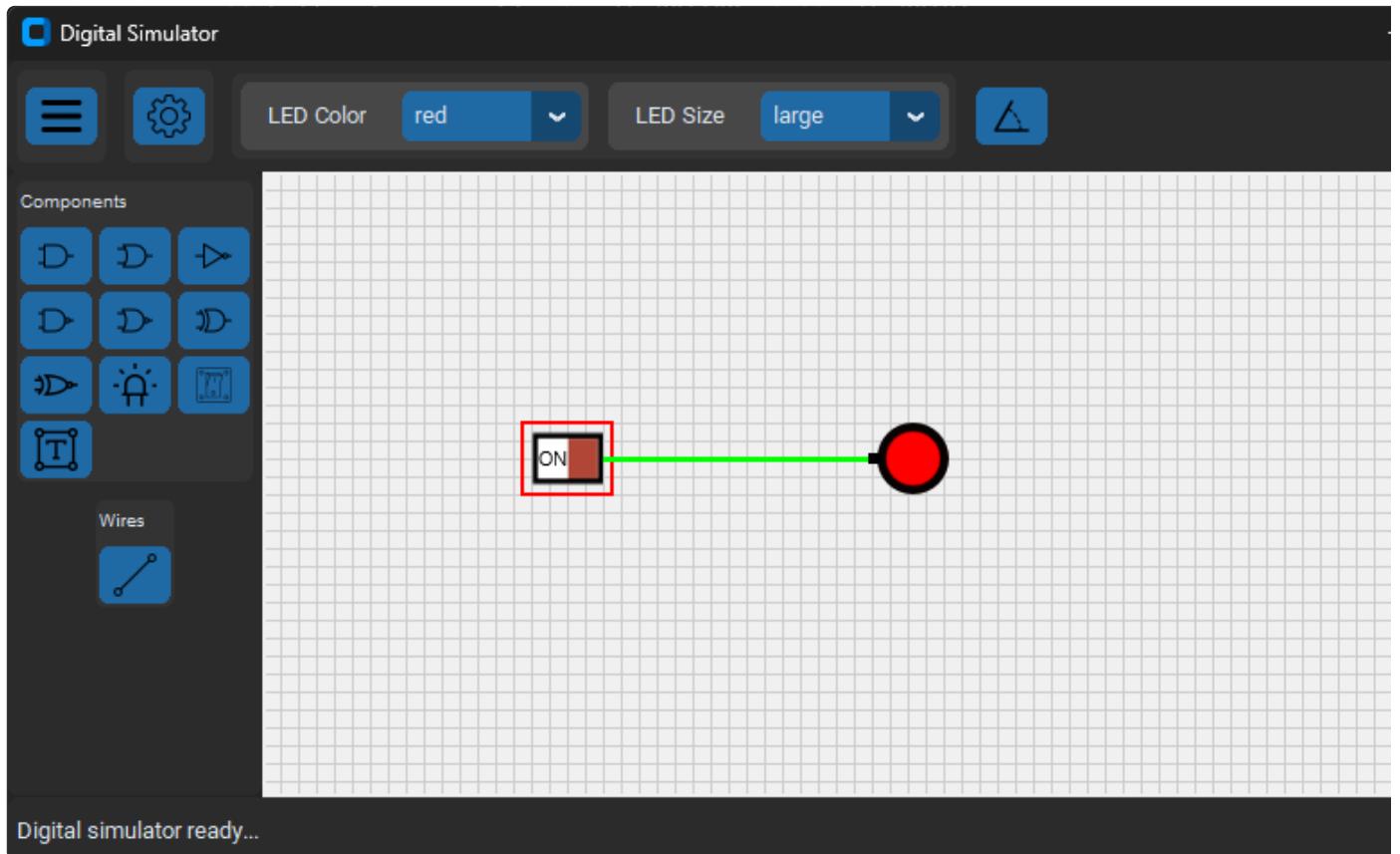
    def set_logic_level(self): # Method to set the led state based on
connected wire state
        if self.wire_list:
            self.state = self.wire_list[0].wire_obj.state

```

JSON Encoder & Decoder

Objectives:

- Create a circuit diagram with a switch, wire, and LED
- Toggle the switch with the right mouse button
- Propagate the logic signal from the switch to the wire to the LED



`digital_simulator.py`

```
...
# Add bindings here
self.bind('<r>', self.rotate_comp)
self.bind("<Configure>", self.on_window_resize)
self.canvas.bind('<Button-3>', self.canvas.edit_shape) # added new
binding to right mouse button

...
```

`UI_Lib/canvas.py`

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid
from Comp_Lib import Switch, Text # Added new import for switch and text
classes

. . .

def edit_shape(self, _event=None): # Added new method
    if isinstance(self.mouse.selected_comp, Switch):
        self.mouse.selected_comp.toggle_state()
        self.redraw()
    elif isinstance(self.mouse.selected_comp, Text):
        pass
```

Comp_Lib/switch.py

```
. . .

def toggle_state(self): # Method to toggle the output state
    self.out_state = not self.out_state
    self.toggle_switch()

    if self.wire_list:
        self.wire_list[0].wire_obj.state = self.out_state
```

Wire.Lib/wire.py

```
from Wire.Lib.wire_selector import WireSelector

class Wire:
    def __init__(self, canvas, x1, y1, x2, y2):
        """Wire base class"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
```

```

# Wire appearance variables
self.fill_color = "black"
self.border_width = 3

self.id = None
self.state = False # Variable to store the wire logic state

. . .

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_wire_color() # Added call to update the wire color
    self.update_selectors()
    self.set_selector_visibility()

. . .

def update_bbox(self):
    self.bbox = self.canvas.bbox(self.id)

def update_wire_color(self): # Method to update wire color based on logic
state
    if self.state:
        self.canvas.itemconfig(self.id, fill="#00ff00")
    else:
        self.canvas.itemconfig(self.id, fill=self.fill_color)

. . .

```

Comp_Lib/led.py

```

. . .

class LED(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "led"
        self.state = False # Variable to store the led logic state
        self.angle = 180

. . .

```

```

def update(self):
    self.set_logic_level() # Call set logic level method to update the led
logic state
    self.update_led_color() # Call to update state
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def update_led_color(self): # Method to update led color based on the led
state
    if self.state:
        self.filename = self.filename_led_on
    else:
        self.filename = self.filename_led_off

. . .

self.move_connected_wires()

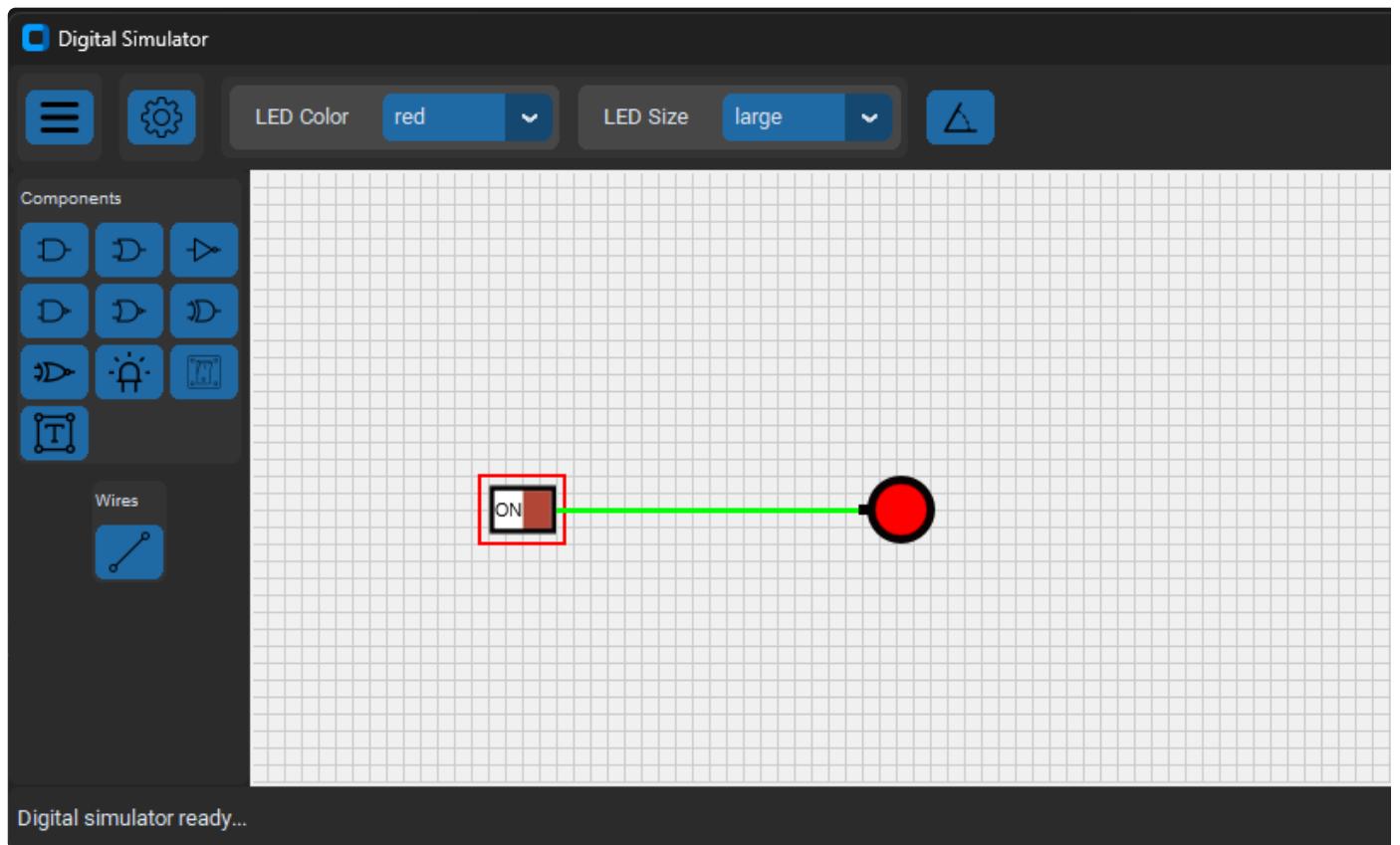
def set_logic_level(self): # Method to set the led state based on
connected wire state
    if self.wire_list:
        self.state = self.wire_list[0].wire_obj.state

```

File Save & Load

Objectives:

- Use custom JSON encoder class to save circuit in a .json file
- Use custom JSON decoder class to load saved circuit
- Verify that the loaded circuit is operational



UI_Lib/file_menu_frame.py

```
import customtkinter as ctk
from tkinter import filedialog as fd
from pathlib import Path
import json # import json library
from PIL import Image

from Comp_Lib import AndGate, OrGate, NandGate, NorGate, XorGate, XnorGate, Switch, LED, Text
from Wire_Lib import Wire, Connection # added import for connection class

class MyEncoder(json.JSONEncoder): # Added custom JSON encoder class
    def default(self, o):
        if hasattr(o, "reprJson"):
            return o.reprJson()
        else:
            return super().default(o)

class JSONDecoder(json.JSONDecoder): # Added custom JSON decoder class
    def __init__(self):
```



```

    dark_image=Image.open
    (Path(__file__).parent /
"../icons/hamburger_menu.png"),
        size=(24, 24))

button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
button.pack(side=ctk.LEFT, padx=5, pady=10)

def new_diagram(self):
    self.canvas.delete("all")
    self.canvas.comp_list = []

def load_diagram(self): # Modified to load json file
try:
    filetypes = (('json files', '*.json'), ('All files', '*.*'))
    f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
    with open(f) as file:
        d = json.load(file)
        print(d)

                # Add components to canvas comp list
for item in d:
    if item['type'] == "wire":
        wire = Wire(self.canvas, x1=item['x1'], y1=item['y1'],
x2=item['x2'], y2=item['y2'])
        self.canvas.comp_list.append(wire)
    elif item['type'] == "switch":
        switch = Switch(self.canvas, x1=item['x1'],
y1=item['y1'])
        switch.angle = item['angle']
        self.canvas.comp_list.append(switch)
    elif item['type'] == "led":
        led = LED(self.canvas, x1=item['x1'], y1=item['y1'])
        led.angle = item['angle']
        self.canvas.comp_list.append(led)

# Add connections to comp wire lists
for item in d:
    if item['type'] == "switch":
        wire_list = item['wire_list']
        for wire_item in wire_list:
            x1 = wire_item['wire_obj']['x1']
            y1 = wire_item['wire_obj']['y1']
            x2 = wire_item['wire_obj']['x2']
            y2 = wire_item['wire_obj']['y2']

```

```

        # Test to see if wire obj matches wire coordinates
        if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
            conn = Connection(switch, wire,
wire_item['wire_end'])
            switch.wire_list.append(conn)

    elif item['type'] == "led":
        wire_list = item['wire_list']
        for wire_item in wire_list:
            x1 = wire_item['wire_obj']['x1']
            y1 = wire_item['wire_obj']['y1']
            x2 = wire_item['wire_obj']['x2']
            y2 = wire_item['wire_obj']['y2']
            # Test to see if wire obj matches wire coordinates
            if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
                conn = Connection(led, wire,
wire_item['wire_end'])
                led.wire_list.append(conn)

    except FileNotFoundError:
        with open('untitled.canvas', 'w') as _file:
            pass
    self.canvas.comp_list = []

    def save_diagram(self): # Modified to save the canvas comp list as a json
file
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
        with open(f, 'w') as file:
            file.write(json.dumps(self.canvas.comp_list, cls=MyEncoder,
indent=4))

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

```

diagram1.json

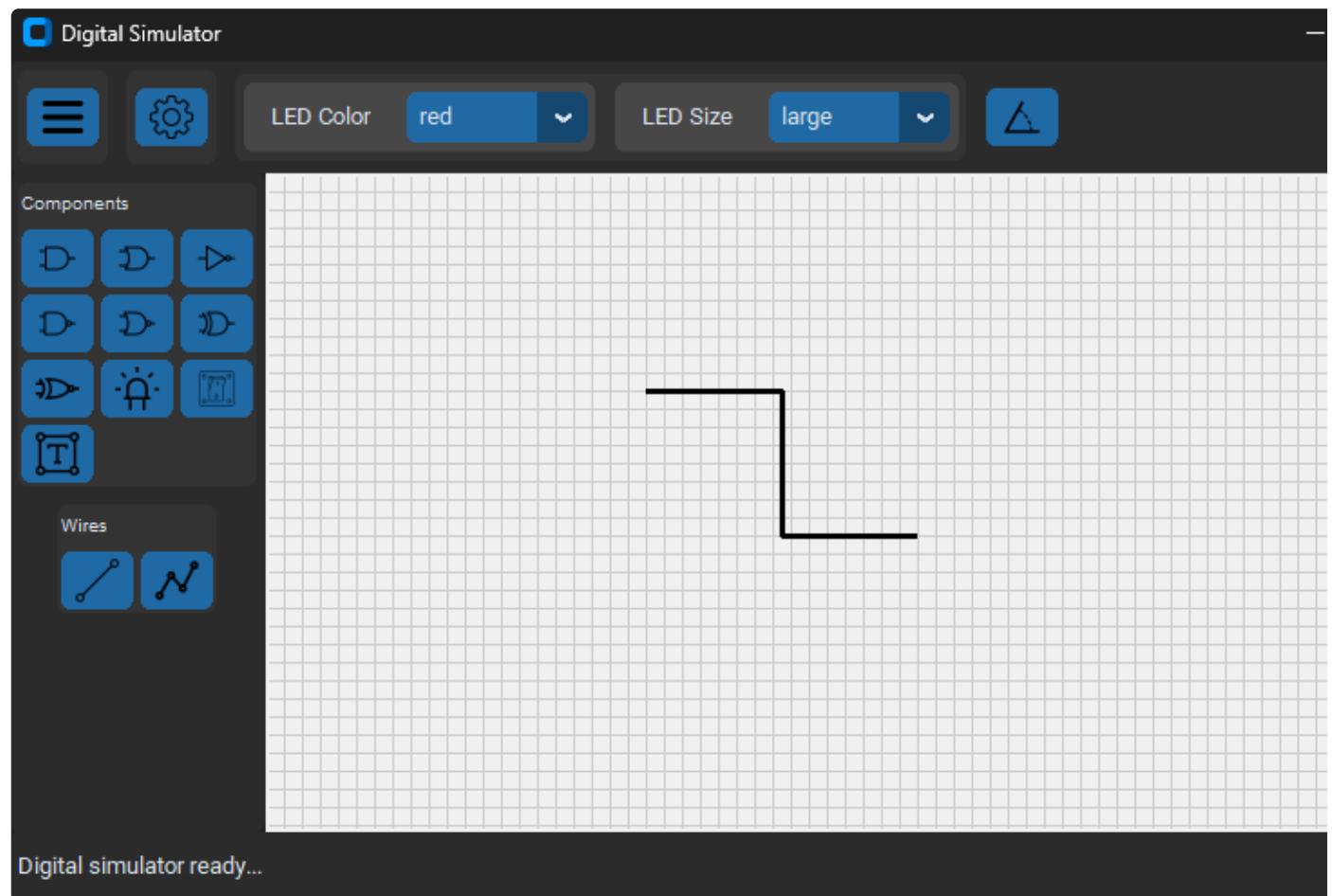
```
[  
  {  
    "type": "switch",  
    "x1": 160,  
    "y1": 200,  
    "angle": 0,  
    "wire_list": [  
      {  
        "type": "connection",  
        "wire_obj": {  
          "type": "wire",  
          "x1": 180.0,  
          "y1": 200.0,  
          "x2": 358.0,  
          "y2": 200.0  
        },  
        "wire_end": "begin"  
      }  
    ]  
  },  
  {  
    "type": "led",  
    "x1": 380,  
    "y1": 200,  
    "angle": 180,  
    "wire_list": [  
      {  
        "type": "connection",  
        "wire_obj": {  
          "type": "wire",  
          "x1": 180.0,  
          "y1": 200.0,  
          "x2": 358.0,  
          "y2": 200.0  
        },  
        "wire_end": "end"  
      }  
    ]  
  },  
  {  
    "type": "wire",  
    "x1": 180.0,  
    "y1": 200.0,  
    "x2": 358.0,  
    "y2": 200.0
```

```
    }  
]
```

Segment Wire Class

Objectives:

- Create a 3-segment wire class
- Configure the segment wire for horizontal or vertical orientation



Wire_Lib/segment_wire.py - new class

```
from Wire_Lib.wire_selector import WireSelector  
  
class SegmentWire:  
    def __init__(self, canvas, x1, y1, x2, y2):  
        """3-Segment Wire"""
```

```

        self.type = "segment_wire"
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        # Wire appearance variables
        self.fill_color = "black"
        self.border_width = 3
        self.line_direction = self.canvas.line_direction

        self.id = None
        self.state = False

        self.bbox = None
        self.is_selected = False
        self.sel_list = []
        self.segment_list = None
        self.points = None
        self.selector = None

        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.create_segmented_line()

        self.s1_id, self.s2_id = None, None
        self.create_selectors()
        self.set_selector_visibility()

    def create_segmented_line(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        segment1, segment2, segment3 = None, None, None

        if self.line_direction == "horizontal":
            segment1 = self.x1, self.y1, self.x1 + w/2, self.y1
            segment2 = self.x1 + w/2, self.y1, self.x1 + w/2, self.y2
            segment3 = self.x1 + w/2, self.y2, self.x2, self.y2
        elif self.line_direction == "vertical":
            segment1 = self.x1, self.y1, self.x1, self.y1 + h/2
            segment2 = self.x1, self.y1 + h/2, self.x2, self.y1 + h/2
            segment3 = self.x2, self.y1 + h/2, self.x2, self.y2
        self.segment_list = [segment1, segment2, segment3]
        self.draw_segments()

    def draw_segments(self):

```

```

        for s in self.segment_list:
            if self.state:
                self.id = self.canvas.create_line(s, fill="#00ff00",
width=self.border_width, tags='wire')
            else:
                self.id = self.canvas.create_line(s, fill=self.fill_color,
width=self.border_width, tags='wire')

    def create_selectors(self):
        """Create selectors at the ends of the wire here"""
        self.s1_id = WireSelector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = WireSelector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_wire_color()
        self.update_selectors()
        self.set_selector_visibility()

    def update_position(self):
        """Update the position when the gate object is moved"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        if self.line_direction == "horizontal":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1 + w / 2,
self.y1,
                               self.x1 + w / 2, self.y1, self.x1 + w / 2,
self.y2,
                               self.x1 + w / 2, self.y2, self.x2, self.y2)
        elif self.line_direction == "vertical":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y1 + h
/ 2,
                               self.x1, self.y1 + h / 2, self.x2, self.y1 + h /
2,
                               self.x2, self.y1 + h / 2, self.x2, self.y2)

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def update_wire_color(self):
        if self.state:
            self.canvas.itemconfig(self.id, fill="#00ff00")
        else:

```

```

        self.canvas.itemconfig(self.id, fill=self.fill_color)

    def update_border_width(self):
        self.canvas.itemconfig(self.id, width=self.border_width)

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

    def set_selector_visibility(self):
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
            # self.x2, self.y2 = self.canvas.grid.snap_to_grid(self.x2,
self.y2)
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
            # self.x1, self.y1 = self.canvas.grid.snap_to_grid(self.x1,
self.y1)

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Wire: " + " x1: " + str(self.x1) + " y1: " + str(self.y1) + \
               " x2: " + str(self.x2) + " y2: " + str(self.y2)

```

```
def reprJson(self):
    return dict(type=self.type, x1=self.x1, y1=self.y1, x2=self.x2,
y2=self.y2)
```

UI_Lib/canvas.py

```
...
class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

        self.led_color = "red"
        self.led_size = "large"
        self.line_direction = "horizontal" # Added variable to store line
direction for segment wires
...
...
```

UI_Lib/wire_button_frame.py

```
import customtkinter as ctk
from pathlib import Path
from PIL import Image

from Wire_Lib import Wire, SegmentWire # Added import for segment wire class


class WireButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.wire = None

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Wires", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)
```

```

    straight_wire_image = ctk.CTkImage(light_image=Image.open # change
variable name to straight wire
                                         (Path(__file__).parent /
"../icons/straight_line.png"),
                                         dark_image=Image.open
                                         (Path(__file__).parent /
"../icons/straight_line.png"),
                                         size=(24, 24))

    straight_wire_button = ctk.CTkButton(self, text="",
image=straight_wire_image, width=30,      # change variable name to straight wire
                                         command=self.create_straight_wire)
    straight_wire_button.grid(row=1, column=0, sticky=ctk.W, padx=2,
pady=2)      # change variable name to straight wire

    segment_wire_image = ctk.CTkImage(light_image=Image.open # Add segment
wire imate
                                         (Path(__file__).parent /
"../icons/segment_line.png"),
                                         dark_image=Image.open
                                         (Path(__file__).parent /
"../icons/segment_line.png"),
                                         size=(24, 24))

    segment_wire_button = ctk.CTkButton(self, text="",
image=segment_wire_image, width=30,  # Add segment wire button
                                         command=self.create_segment_wire)
    segment_wire_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
# Add segment wire button

# Shape button handlers
def create_straight_wire(self):
    wire = Wire(self.canvas, 0, 0, 0, 0)
    self.canvas.comp_list.append(wire)
    self.canvas.mouse.current_wire_obj = wire
    self.canvas.show_connectors()
    self.canvas.mouse.draw_wire_mouse_events()

def create_segment_wire(self): # Added method to create segment wire
    print("Create segment wire called")
    wire = SegmentWire(self.canvas, 0, 0, 0, 0)
    self.canvas.comp_list.append(wire)
    self.canvas.mouse.current_wire_obj = wire
    self.canvas.show_connectors()

```

```
self.canvas.mouse.draw_wire_mouse_events()
```

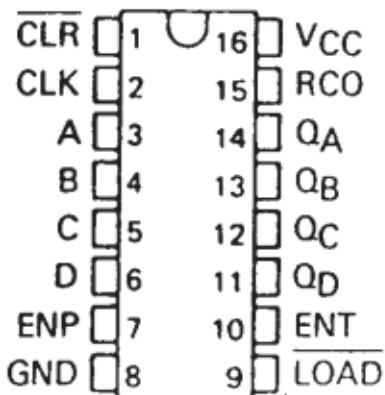
74ls161 Counter IC Class

Objectives:

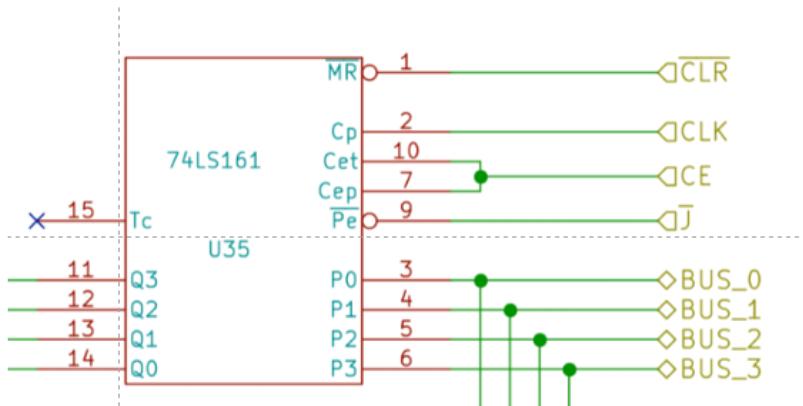
- Get the pin-out and logic diagram for 74ls161 IC
- Draw the image of the IC
- Create a 74161 IC class
- Add an IC frame to the Left Frame Class
- Create the logic for the 74161 counter

[74ls161 Data Sheet](#)

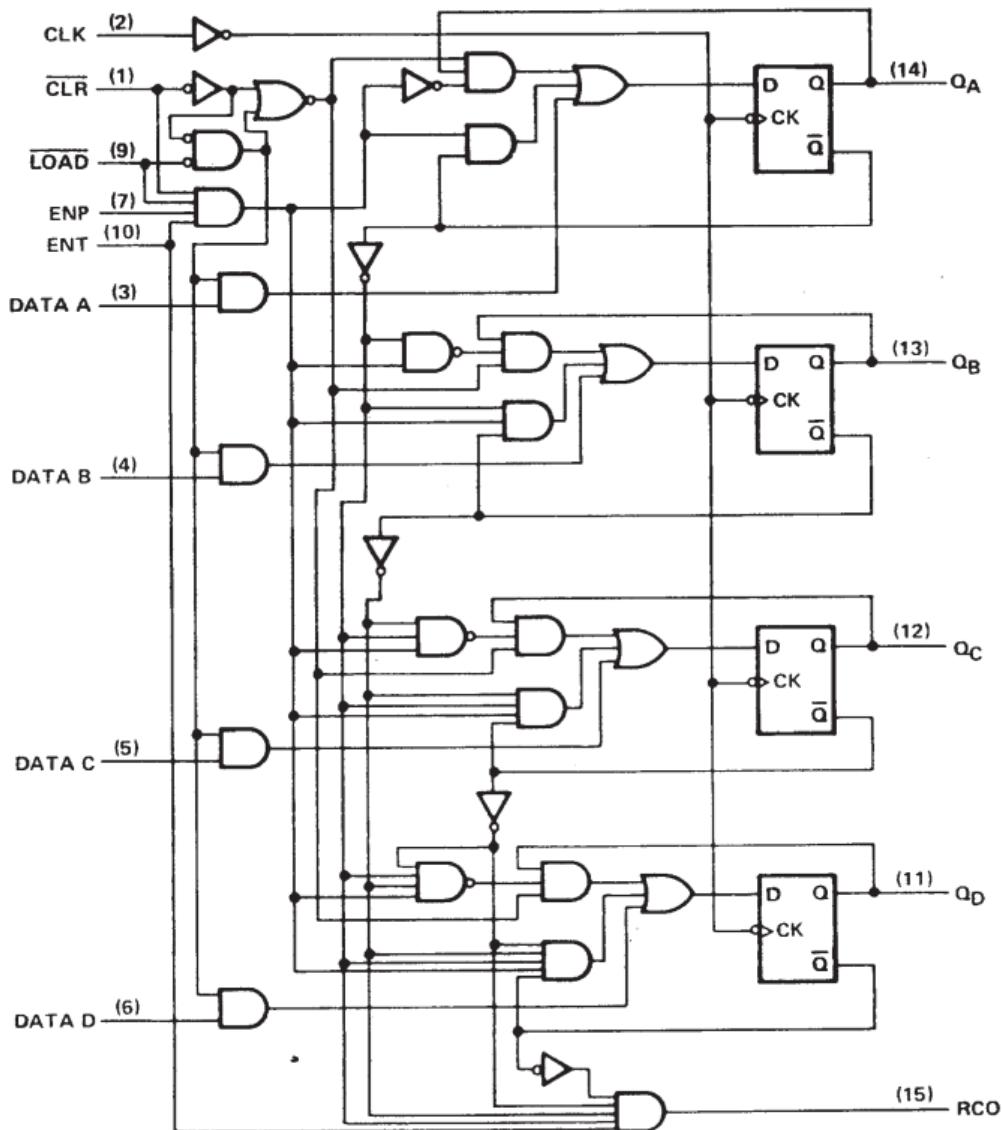
Device pin-out



Device alternate pin-out for simulation

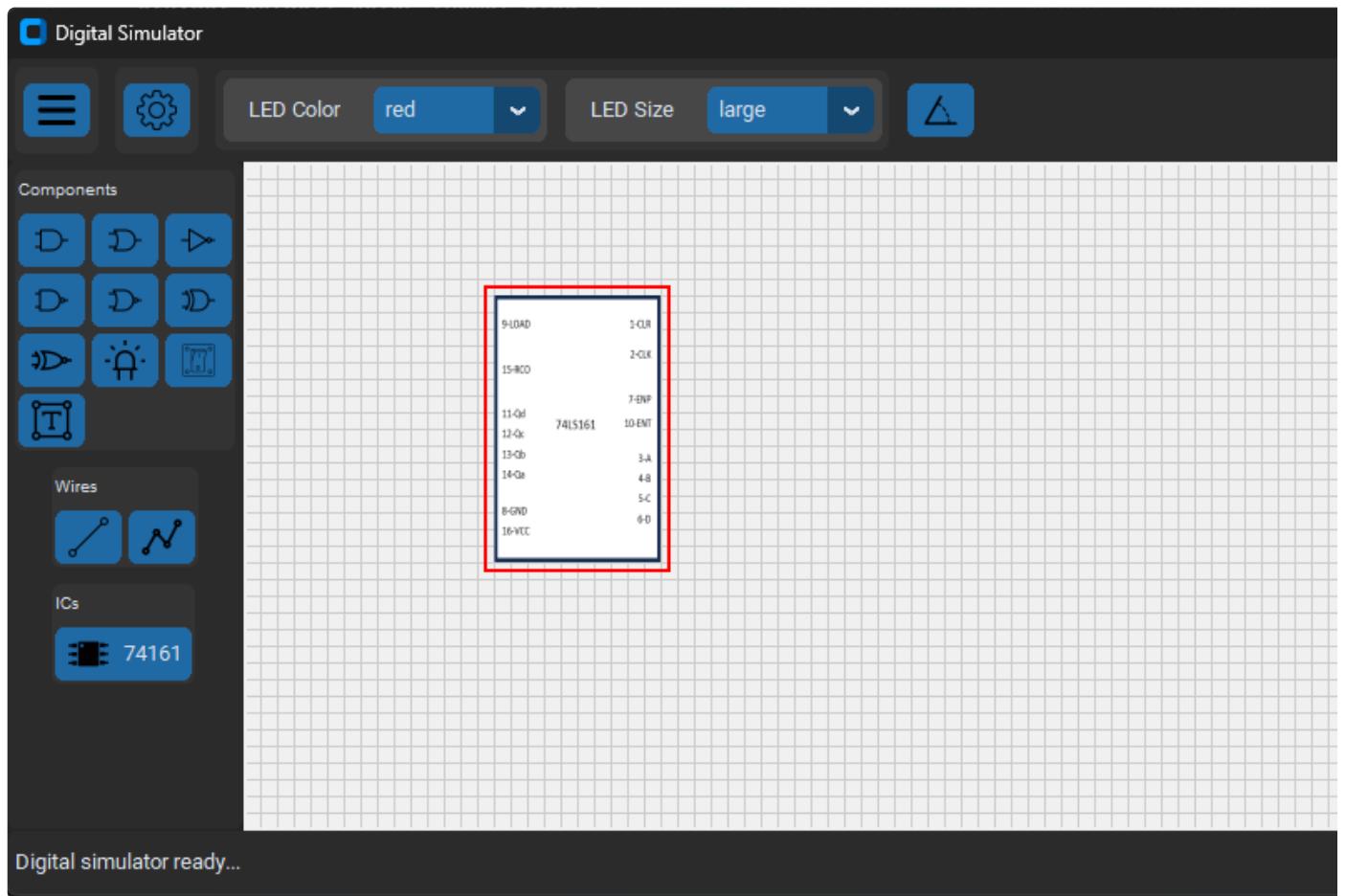


Device logic diagram



Notes:

- Data A, B, C, D (pins 3, 4, 5, 6) - input data to set counter to known value if $|LOAD$ (pin 9) is enabled (set to False)
- Q_a , Q_b , Q_c , Q_d (pins 14, 13, 12, 11) - output the current counter data
- ENP, ENT (pins 7, 10) - Chip enable
- CLK (pin 2) - Clock input
- $|CLR$ (pin 1) - Clear input (enabled when set to False)
- RCO (pin 15) - Not used
- GND (Pin 8) - power ground
- VCC (Pin 16) - power voltage



74LS161 Counter Image

- Drawn in PowerPoint
- Copied to Paint3D
- Resized to w=100 and h=160

940AD		140R
		240E
1540C		
		740P
114d	74LS161	10-BMT
134c		3-A
144a		4-B
		5-C
845D		6-D
164TC		

IC_Lib/IC.py - IC base class - new class

```

import tkinter as tk
from PIL import Image, ImageTk

class IC:
    def __init__(self, canvas, x1, y1):
        """Base class for integrated circuit (IC) classes"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0
        self.filename = None

        self.is_selected = False
        self.is_drawing = False

        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.wire_list = []

```

```

def create_image(self, filename):
    """Initial component image creation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True)
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

def update_position(self):
    """Update the position when the gate object is moved"""
    self.canvas.coords(self.id, self.x1, self.y1) # Update position

def update_image(self, filename):
    """Update the image for gate symbol rotation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True) # Update
image rotation
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.canvas.itemconfig(self.id, image=self.ph_image) # Update image

def update_bbox(self):
    """Update the bounding box to get current gate coordinates"""
    self.bbox = self.canvas.bbox(self.id)

def create_selector(self):
    """Create the red rectangle selector and check to see if the gate is
selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
    self.set_selector_visibility()

def update_selector(self):
    """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.canvas.coords(self.sel_id, x1, y1, x2, y2)
    self.set_selector_visibility()

def set_selector_visibility(self):
    """Set the selector visibility state"""
    if self.is_selected:
        self.canvas.itemconfig(self.sel_id, state='normal')
    else:

```

```

        self.canvas.itemconfig(self.sel_id, state='hidden')

    def set_connector_visibility(self):
        """Set the connector visibility state"""
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

    def rotate(self):
        """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
        if angle > 270 deg"""
        self.angle += 90
        if self.angle > 270:
            self.angle = 0

    def check_connector_hit(self, x, y):
        """Hit test to see if a connector is at the provided x, y
        coordinates"""
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None

    def move_connected_wires(self):
        """Resize connected wires if the shape is moved"""
        for connection in self.wire_list:
            for connector in self.conn_list:
                if connector == connection.connector_obj:
                    # print(connector, connection.line_obj, "Match")
                    if connection.wire_end == "begin":
                        connection.wire_obj.x1 = connector.x
                        connection.wire_obj.y1 = connector.y
                    elif connection.wire_end == "end":
                        connection.wire_obj.x2 = connector.x
                        connection.wire_obj.y2 = connector.y

```

IC_Lib/ic74161_counter.py

```
from pathlib import Path
```

```

from IC_Lib.ic import IC

class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.filename = Path(__file__).parent /
"../images/ics/74161_easy_100x160.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        # self.create_connectors()
        # self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        # self.update_connectors()
        # self.set_connector_visibility()

```

UI_Lib/ic_button_frame.py - new class

```

import customtkinter as ctk
from pathlib import Path
from PIL import Image

from IC_Lib import IC74161


class ICButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="ICs", font=("Helvetica",
10), height=20)

```

```

        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
                               padx=2, pady=2)

        ic_image = ctk.CTkImage(light_image=Image.open
                                (Path(__file__).parent / "../icons/ic.png"),
                                dark_image=Image.open
                                (Path(__file__).parent / "../icons/ic.png"),
                                size=(24, 24))

        ic_button = ctk.CTkButton(self, text="74161", image=ic_image, width=30,
                                  command=self.create_ic_74161)
        ic_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)

    def create_ic_74161(self):
        ic = IC74161(self.canvas, 100, 100)
        self.canvas.comp_list.append(ic)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()

```

UI_Lib/left_frame.py

```

import customtkinter as ctk
from UI_Lib.comp_button_frame import CompButtonFrame
from UI_Lib.wire_button_frame import WireButtonFrame
from UI_Lib.ic_button_frame import ICBButtonFrame


class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

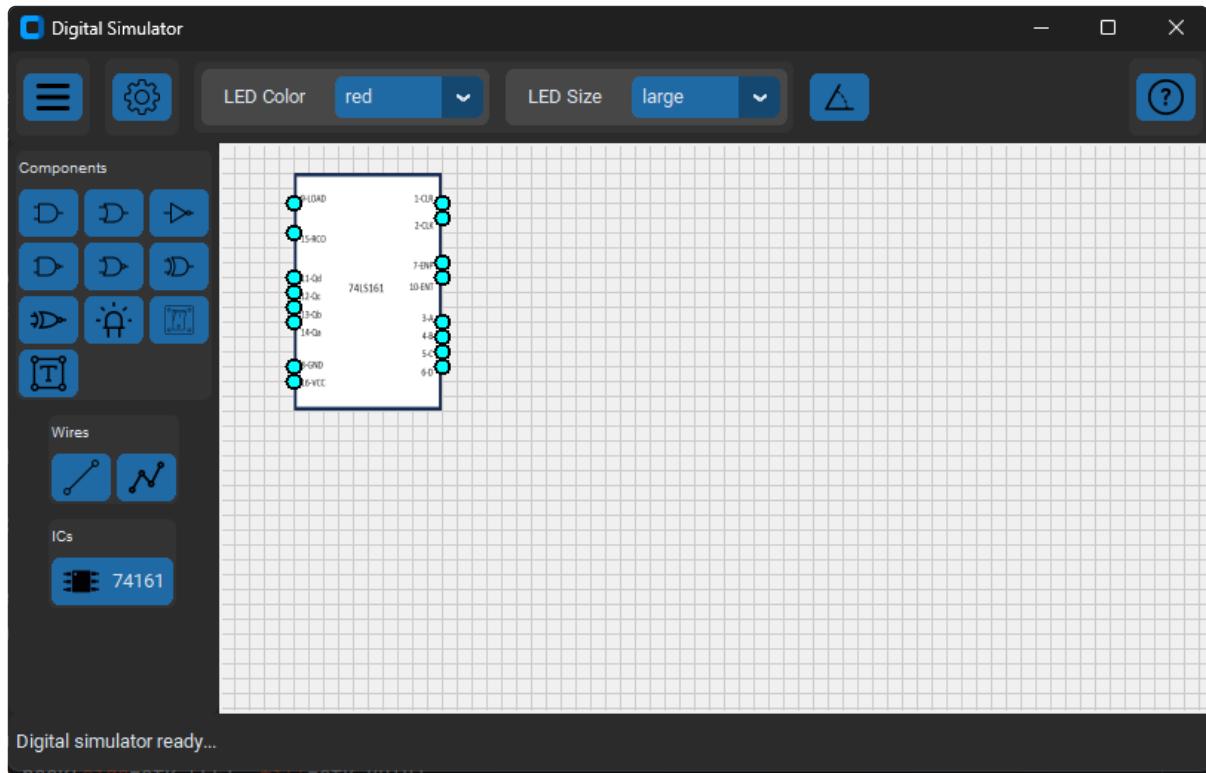
        self.comp_button_frame = CompButtonFrame(self, self.canvas)
        self.comp_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.wire_button_frame = WireButtonFrame(self, self.canvas)
        self.wire_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.ic_button_frame = ICBButtonFrame(self, self.canvas)
        self.ic_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

```

74LS161 IC Connectors



IC.Lib/ic74161_counter.py

```
from pathlib import Path

from IC.Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.logic_dict = {}

        # Set initial logic states
        for i in range(1, 17): # 16 pin IC
            self.logic_dict['c' + str(i)] = False
        self.logic_dict['c14'] = True

        self.filename = Path(__file__).parent /
```

```
    ".../images/ics/74161_easy_100x160.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors() # Added call to create connectors
        self.set_connector_visibility() # Added call to set connector
visibility

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors() # Added call to update connectors
        self.set_connector_visibility() # Added call to set connector
visibility

    def create_connectors(self): # Added new method to create connectors
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.conn_list.append(Connector(self.canvas, "c1", center.x + w/2,
center.y - h/2 + 20))
        self.conn_list.append(Connector(self.canvas, "c2", center.x + w / 2,
center.y - h / 2 + 30))

        self.conn_list.append(Connector(self.canvas, "c3", center.x + w / 2,
center.y - h / 2 + 100))
        self.conn_list.append(Connector(self.canvas, "c4", center.x + w / 2,
center.y - h / 2 + 110))
        self.conn_list.append(Connector(self.canvas, "c5", center.x + w / 2,
center.y - h / 2 + 120))
        self.conn_list.append(Connector(self.canvas, "c6", center.x + w / 2,
center.y - h / 2 + 130))

        self.conn_list.append(Connector(self.canvas, "c7", center.x + w / 2,
center.y - h / 2 + 60))
        self.conn_list.append(Connector(self.canvas, "c10", center.x + w / 2,
center.y - h / 2 + 70))

        self.conn_list.append(Connector(self.canvas, "c9", center.x - w / 2,
center.y - h/2 + 20))
        self.conn_list.append(Connector(self.canvas, "c15", center.x - w / 2,
center.y - h / 2 + 40))
```

```

        self.conn_list.append(Connector(self.canvas, "c11", center.x - w / 2,
center.y - h / 2 + 70))
        self.conn_list.append(Connector(self.canvas, "c12", center.x - w / 2,
center.y - h / 2 + 80))
        self.conn_list.append(Connector(self.canvas, "c13", center.x - w / 2,
center.y - h / 2 + 90))
        self.conn_list.append(Connector(self.canvas, "c14", center.x - w / 2,
center.y - h / 2 + 100))
        self.conn_list.append(Connector(self.canvas, "c8", center.x - w / 2,
center.y - h / 2 + 130))
        self.conn_list.append(Connector(self.canvas, "c16", center.x - w / 2,
center.y - h / 2 + 140))

    def update_connectors(self): # Added new method to update connectors
        # Recalculate position of connectors from current shape position and
size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.conn_list[0].x, self.conn_list[0].y = center.x + w/2, center.y -
h/2 + 20
        self.conn_list[1].x, self.conn_list[1].y = center.x + w / 2, center.y -
h / 2 + 30
        self.conn_list[2].x, self.conn_list[2].y = center.x + w / 2, center.y -
h / 2 + 100
        self.conn_list[3].x, self.conn_list[3].y = center.x + w / 2, center.y -
h / 2 + 110
        self.conn_list[4].x, self.conn_list[4].y = center.x + w / 2, center.y -
h / 2 + 120
        self.conn_list[5].x, self.conn_list[5].y = center.x + w / 2, center.y -
h / 2 + 130
        self.conn_list[6].x, self.conn_list[6].y = center.x + w / 2, center.y -
h / 2 + 60
        self.conn_list[7].x, self.conn_list[7].y = center.x + w / 2, center.y -
h / 2 + 70

        self.conn_list[8].x, self.conn_list[8].y = center.x - w/2, center.y -
h/2 + 20
        self.conn_list[9].x, self.conn_list[9].y = center.x - w / 2, center.y -
h / 2 + 40
        self.conn_list[10].x, self.conn_list[10].y = center.x - w / 2, center.y -
h / 2 + 70
        self.conn_list[11].x, self.conn_list[11].y = center.x - w / 2, center.y -
h / 2 + 80
        self.conn_list[12].x, self.conn_list[12].y = center.x - w / 2, center.y

```

```

- h / 2 + 90
    self.conn_list[13].x, self.conn_list[13].y = center.x - w / 2, center.y
- h / 2 + 100
    self.conn_list[14].x, self.conn_list[14].y = center.x - w / 2, center.y
- h / 2 + 130
    self.conn_list[15].x, self.conn_list[15].y = center.x - w / 2, center.y
- h / 2 + 140

    # Draw the connectors
    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def set_logic_level(self):
    # c1 = |CLR
    # c2 = CLK
    # c3 = A
    # c4 = B
    # c5 = C
    # c6 = D
    # c7 = ENP
    # c8 = GND
    # c9 = |LOAD
    # c10 = ENT
    # c11 = Qd
    # c12 = Qc
    # c13 = Qd
    # c14 = Qa
    # C15 = RCO
    # C16 = VCC

    for wire in self.wire_list:
        # NAND Gate #1
        if wire.connector_obj.name == "c1":
            self.logic_dict['c1'] = wire.line_obj.state
            self.logic_dict['c3'] = not(self.logic_dict['c1'] and
self.logic_dict['c2'])
        elif wire.connector_obj.name == "c2":
            self.logic_dict['c2'] = wire.line_obj.state
            self.logic_dict['c3'] = not(self.logic_dict['c1'] and
self.logic_dict['c2']))
        elif wire.connector_obj.name == "c3":
            wire.line_obj.state = self.logic_dict['c3']

    # NAND Gate #2

```

```

        elif wire.connector_obj.name == "c4":
            self.logic_dict['c4'] = wire.line_obj.state
            self.logic_dict['c6'] = not(self.logic_dict['c4'] and
self.logic_dict['c5'])
        elif wire.connector_obj.name == "c5":
            self.logic_dict['c5'] = wire.line_obj.state
            self.logic_dict['c6'] = not(self.logic_dict['c4'] and
self.logic_dict['c5'])
        elif wire.connector_obj.name == "c6":
            wire.line_obj.state = self.logic_dict['c6']

# NAND Gate #3
elif wire.connector_obj.name == "c10":
    self.logic_dict['c10'] = wire.line_obj.state
    self.logic_dict['c8'] = not(self.logic_dict['c10'] and
self.logic_dict['c9'])
elif wire.connector_obj.name == "c9":
    self.logic_dict['c9'] = wire.line_obj.state
    self.logic_dict['c8'] = not(self.logic_dict['c10'] and
self.logic_dict['c9'])
elif wire.connector_obj.name == "c8":
    wire.line_obj.state = self.logic_dict['c8']

# NAND Gate #4
elif wire.connector_obj.name == "c13":
    self.logic_dict['c13'] = wire.line_obj.state
    self.logic_dict['c11'] = not(self.logic_dict['c13'] and
self.logic_dict['c12'])
elif wire.connector_obj.name == "c12":
    self.logic_dict['c12'] = wire.line_obj.state
    self.logic_dict['c11'] = not(self.logic_dict['c13'] and
self.logic_dict['c12'])
elif wire.connector_obj.name == "c11":
    wire.line_obj.state = self.logic_dict['c11']

```

74LS161 Logic Levels

Start by testing a binary counter
Sandbox/test_binary_count.py

```

def add_one(num):
    num += 1
    if num > 15:
        num = 0
    return num

def parse_binary(binary):
    d1 = binary[3]
    d2 = binary[2]
    d3 = binary[1]
    d4 = binary[0]
    print("d1: ", d1, " d2: ", d2, " d3: ", d3, " d4: ", d4)

num_int = 0
for i in range(1, 16):
    num_int = add_one(num_int)
    binary = format(num_int, '04b')
    print(binary)
    parse_binary(binary)

```

Console Output:

```

0001
d1: 1 d2: 0 d3: 0 d4: 0
0010
d1: 0 d2: 1 d3: 0 d4: 0
0011
d1: 1 d2: 1 d3: 0 d4: 0
0100
d1: 0 d2: 0 d3: 1 d4: 0
0101
d1: 1 d2: 0 d3: 1 d4: 0
0110
d1: 0 d2: 1 d3: 1 d4: 0
0111
d1: 1 d2: 1 d3: 1 d4: 0
1000
d1: 0 d2: 0 d3: 0 d4: 1

```

```

1001
d1: 1 d2: 0 d3: 0 d4: 1

1010
d1: 0 d2: 1 d3: 0 d4: 1

1011
d1: 1 d2: 1 d3: 0 d4: 1

1100
d1: 0 d2: 0 d3: 1 d4: 1

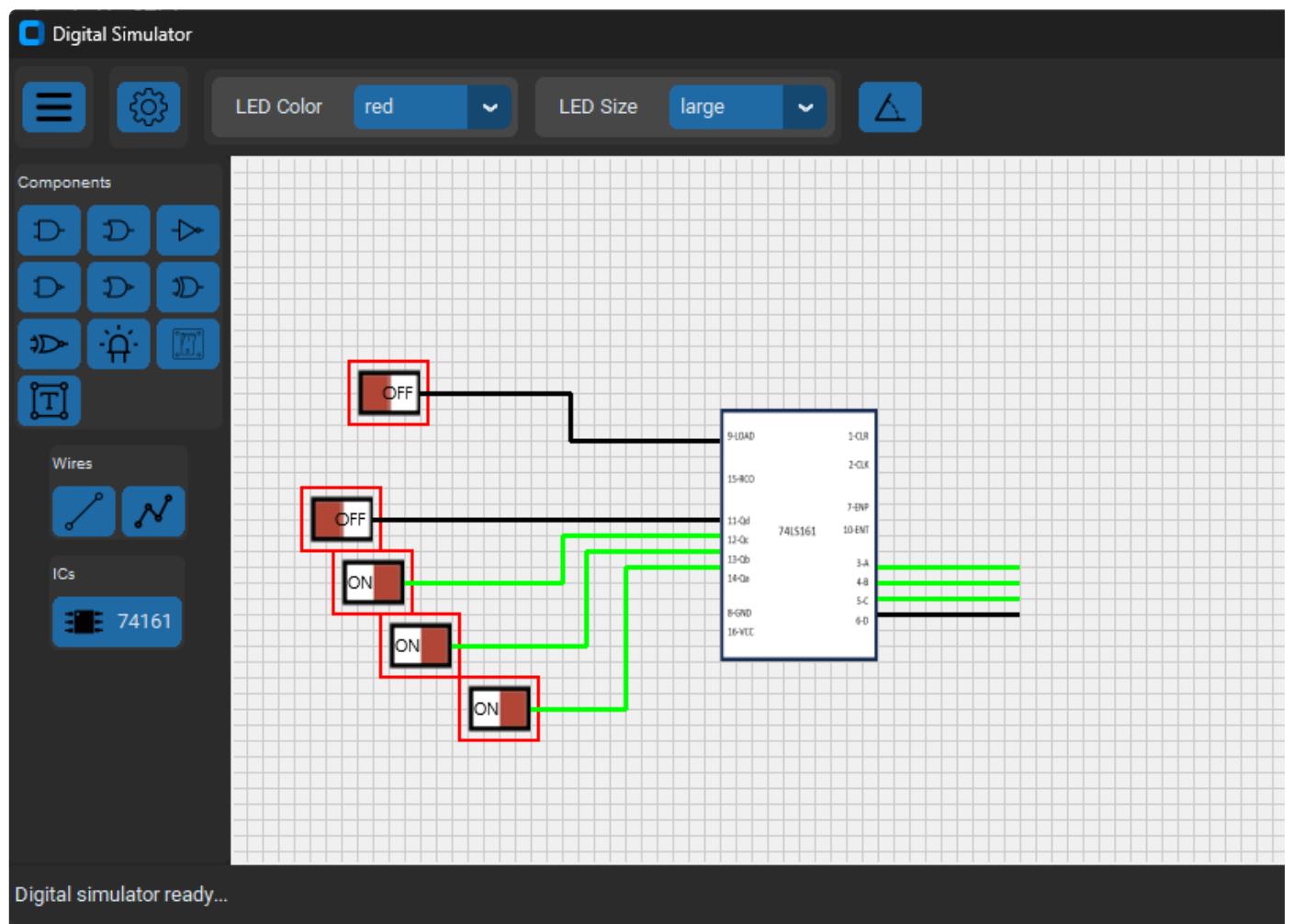
1101
d1: 1 d2: 0 d3: 1 d4: 1

1110
d1: 0 d2: 1 d3: 1 d4: 1

1111
d1: 1 d2: 1 d3: 1 d4: 1

```

Load Test



Test: Good

IC_Lib/ic74161_counter.py

```
    . . .

class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.logic_dict = {}
        self.count_int = 0 # Added a variable to store counter value as
integer

        # Initialize 4 x D-flip flops
        self.d1, self.d2, self.d3, self.d4 = False, False, False, False # Initialize the 4 x D-flip flops

        # Set initial logic states
        for i in range(1, 17): # 16 pin IC
            self.logic_dict['c' + str(i)] = False

    . . .

    def update(self):
        self.set_logic_level() # Added call to set logic level
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    . . .

    def set_logic_level(self): # Added new method to set logic levels
        # c1 = |CLR
        # c2 = CLK
        # c3 = A
        # c4 = B
        # c5 = C
        # c6 = D
        # c7 = ENP
        # c8 = GND
        # c9 = |LOAD
```

```

# c10 = ENT
# c11 = Qd
# c12 = Qc
# c13 = Qd
# c14 = Qa
# C15 = RCO
# C16 = VCC

for wire in self.wire_list:
    # NAND Gate #1
    if wire.connector_obj.name == "c1": # CLR input
        self.logic_dict['c1'] = wire.wire_obj.state
        if not wire.wire_obj.state:
            self.d1 = self.d2 = self.d3 = self.d4 = False
            self.set_ABCD()
    elif wire.connector_obj.name == "c2": # CLK input
        self.logic_dict['c2'] = wire.wire_obj.state
        if wire.wire_obj.state:
            self.count_int = self.add_one(self.count_int)
            binary = format(self.count_int, '04b')
            self.parse_binary(binary)
            self.set_ABCD()
    elif wire.connector_obj.name == "c3": # Output A
        wire.wire_obj.state = self.logic_dict['c3']
    elif wire.connector_obj.name == "c4": # Output B
        wire.wire_obj.state = self.logic_dict['c4']
    elif wire.connector_obj.name == "c5": # Output C
        wire.wire_obj.state = self.logic_dict['c5']
    elif wire.connector_obj.name == "c6": # Output D
        wire.wire_obj.state = self.logic_dict['c6']
    elif wire.connector_obj.name == "c7": # ENP - enable input
        self.logic_dict['c7'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c10": # ENT - enable input
        self.logic_dict['c10'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c8": # GND - input
        self.logic_dict['c8'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c16": # VCC - input
        self.logic_dict['c16'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c9": # | LOAD - input
        self.logic_dict['c9'] = wire.wire_obj.state
        if not wire.wire_obj.state:
            self.d1 = self.logic_dict["c14"]
            self.d2 = self.logic_dict["c13"]
            self.d3 = self.logic_dict["c12"]
            self.d4 = self.logic_dict["c11"]
            self.set_ABCD()

```

```

        elif wire.connector_obj.name == "c11": # Qd - input
            self.logic_dict['c11'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c12": # Qc - input
            self.logic_dict['c12'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c13": # Qb - input
            self.logic_dict['c13'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c14": # Qa - input
            self.logic_dict['c14'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c15": # RCO - input
            self.logic_dict['c15'] = wire.wire_obj.state

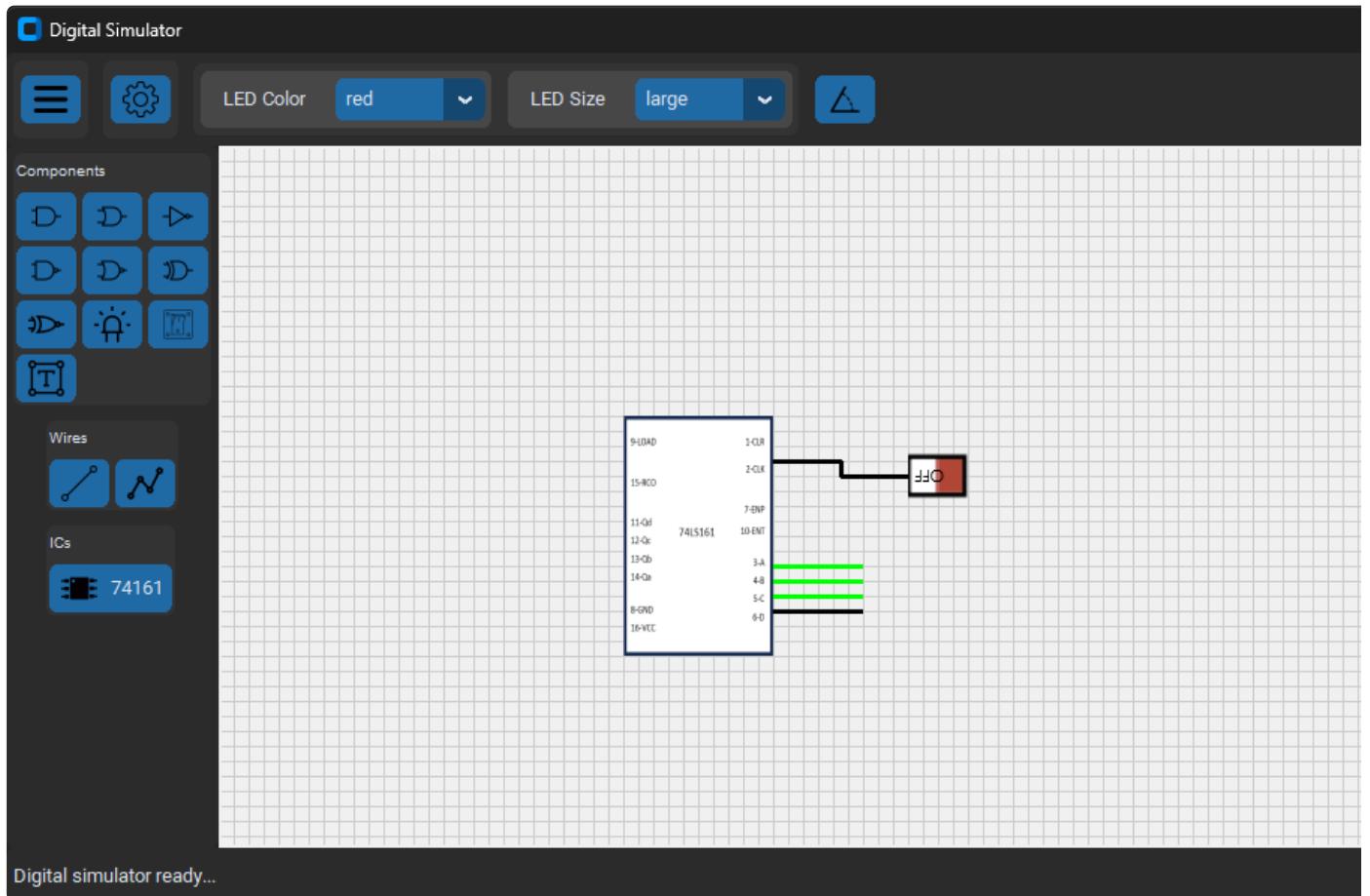
    def set_ABCD(self): # Helper method to set the IC outputs
        self.logic_dict['c3'] = self.d1
        self.logic_dict['c4'] = self.d2
        self.logic_dict['c5'] = self.d3
        self.logic_dict['c6'] = self.d4

    @staticmethod
    def add_one(num): # Helper method to add numbers up to 15 then reset to 0
        num += 1
        if num > 15:
            num = 0
        return num

    def parse_binary(self, bin_num): # Helper method to set D flip-flops to a
        4-bit binary number
        self.d1 = bin_num[3]
        self.d2 = bin_num[2]
        self.d3 = bin_num[1]
        self.d4 = bin_num[0]

```

Clock Test



Watch out for binary 1/0 to True/False mismatches.

IC.Lib/ic74161_counter.py

```
from pathlib import Path

from IC.Lib.ic import IC
from Helper.Lib.point import Point
from Wire.Lib.connector import Connector


class IC74161(IC):
    """Model for 74ls161 Counter IC - 16-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74161"

        self.logic_dict = {}
        self.count_int = 0 # Added variable to store count

        # Initialize 4 x D-flip flops
```

```

    self.d1, self.d2, self.d3, self.d4 = False, False, False, False # Initialize the 4xD flip-flops

    # Set initial logic states
    for i in range(1, 17): # 16 pin IC
        self.logic_dict['c' + str(i)] = False

    self.filename = Path(__file__).parent /
"../images/ics/74161_easy_100x160.png"
    self.create_image(self.filename)
    self.update_bbox()
    self.create_selector()
    self.create_connectors()
    self.set_connector_visibility()

    . . .

def set_logic_level(self):
    # c1 = |CLR
    # c2 = CLK
    # c3 = A
    # c4 = B
    # c5 = C
    # c6 = D
    # c7 = ENP
    # c8 = GND
    # c9 = |LOAD
    # c10 = ENT
    # c11 = Qd
    # c12 = Qc
    # c13 = Qd
    # c14 = Qa
    # C15 = RCO
    # C16 = VCC

    for wire in self.wire_list:
        # print(wire)
        if wire.connector_obj.name == "c1": # CLR input
            self.logic_dict['c1'] = wire.wire_obj.state
            if not wire.wire_obj.state:
                self.d1 = self.d2 = self.d3 = self.d4 = False
                self.set_ABCD()
        elif wire.connector_obj.name == "c2": # CLK input
            self.logic_dict['c2'] = wire.wire_obj.state
            if wire.wire_obj.state:
                self.count_int = self.add_one(self.count_int)

```

```

        binary = format(self.count_int, '04b')
        self.parse_binary(binary)
        self.set_ABCD()

    elif wire.connector_obj.name == "c3": # Output A
        wire.wire_obj.state = self.logic_dict['c3']
        # print("c3 wire: ", wire.wire_obj, " state ",
        wire.wire_obj.state)

    elif wire.connector_obj.name == "c4": # Output B
        wire.wire_obj.state = self.logic_dict['c4']
        # print("c4 wire: ", wire.wire_obj, " state ",
        wire.wire_obj.state)

    elif wire.connector_obj.name == "c5": # Output C
        wire.wire_obj.state = self.logic_dict['c5']
        # print("c5 wire: ", wire.wire_obj, " state ",
        wire.wire_obj.state)

    elif wire.connector_obj.name == "c6": # Output D
        wire.wire_obj.state = self.logic_dict['c6']
        # print("c6 wire: ", wire.wire_obj, " state ",
        wire.wire_obj.state)

    elif wire.connector_obj.name == "c7": # ENP - enable input
        self.logic_dict['c7'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c10": # ENT - enable input
        self.logic_dict['c10'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c8": # GND - input
        self.logic_dict['c8'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c16": # VCC - input
        self.logic_dict['c16'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c9": # |LOAD - input
        self.logic_dict['c9'] = wire.wire_obj.state

    if not wire.wire_obj.state:
        self.d1 = self.logic_dict["c14"]
        self.d2 = self.logic_dict["c13"]
        self.d3 = self.logic_dict["c12"]
        self.d4 = self.logic_dict["c11"]
        self.set_ABCD()

    elif wire.connector_obj.name == "c11": # Qd - input
        self.logic_dict['c11'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c12": # Qc - input
        self.logic_dict['c12'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c13": # Qb - input
        self.logic_dict['c13'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c14": # Qa - input
        self.logic_dict['c14'] = wire.wire_obj.state

    elif wire.connector_obj.name == "c15": # RCO - input
        self.logic_dict['c15'] = wire.wire_obj.state

```

```

def set_ABCD(self):
    self.logic_dict['c3'] = self.d1
    self.logic_dict['c4'] = self.d2
    self.logic_dict['c5'] = self.d3
    self.logic_dict['c6'] = self.d4

    @staticmethod
    def add_one(num):
        num += 1
        if num > 15:
            num = 0
        return num

    def parse_binary(self, bin_num):
        def parse_binary(self, bin_num):
            result = lambda s: True if s == '1' else False
            self.d1 = result(bin_num[3])
            self.d2 = result(bin_num[2])
            self.d3 = result(bin_num[1])
            self.d4 = result(bin_num[0])

```

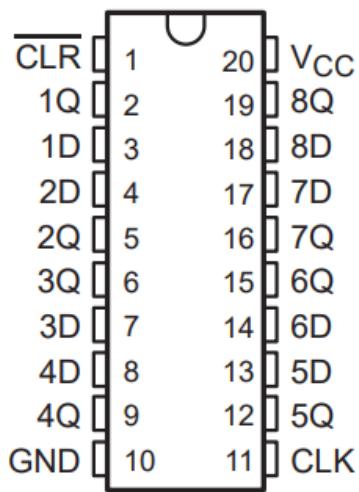
74ls273 Octal D Flip-Flops Class

Objectives:

- Create a new 74273 IC class
- Create a new 74273 image file with "Ds" on one side and "Qs" on the other side
- Test the IC as a storage register

[74LS273 Data Sheet](#)

Device pin-out

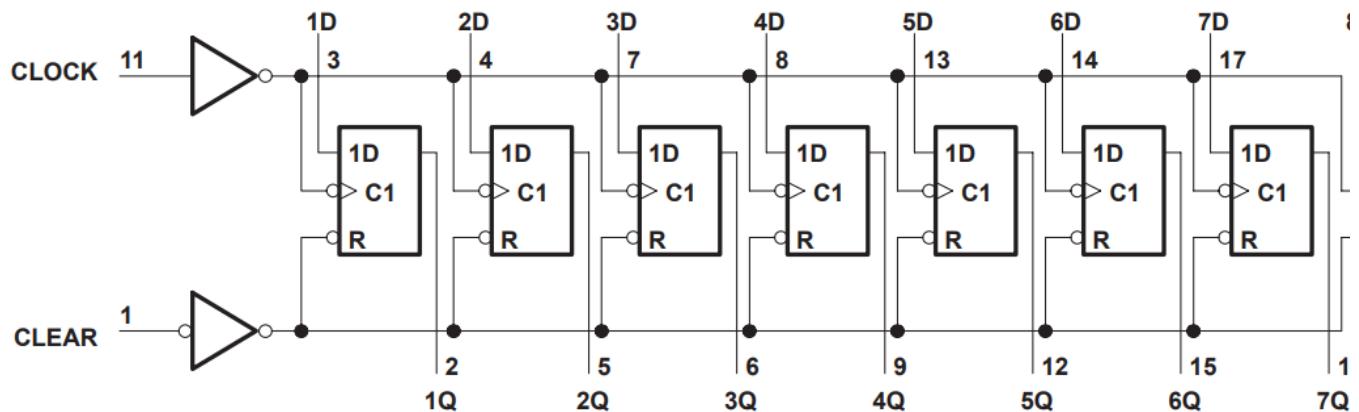


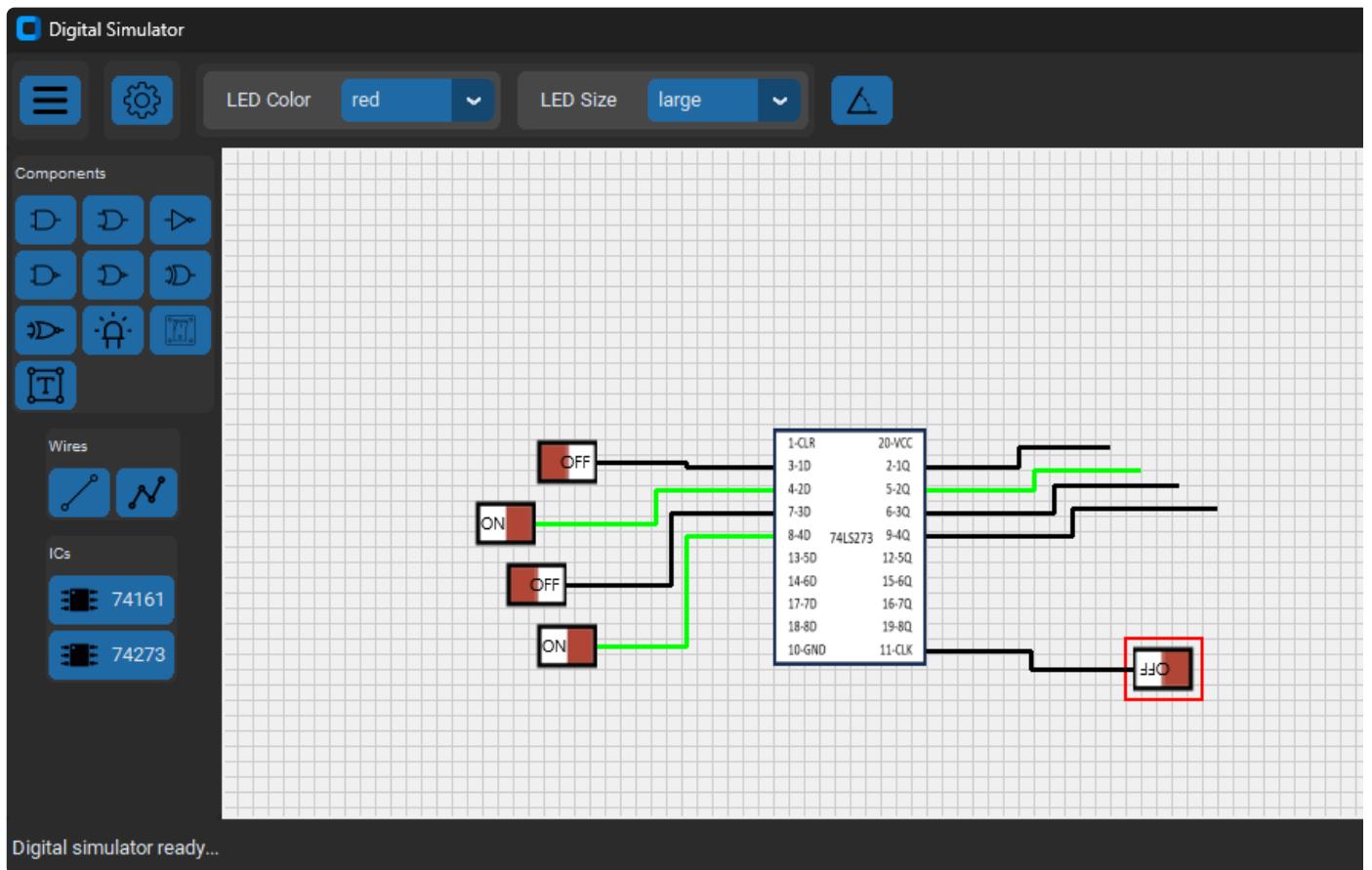
Function Table

FUNCTION TABLE
(each flip-flop)

INPUTS			OUTPUT Q
CLEAR	CLOCK	D	
L	X	X	L
H	↑	H	H
H	↑	L	L
H	L	X	Q ₀

Logic Diagram





IC.Lib/ic_74273_flip_flop.py

```
from pathlib import Path

from IC.Lib.ic import IC
from Helper.Lib.point import Point
from Wire.Lib.connector import Connector


class DFlipFlop:
    """Logical model for D Flip-Flop"""
    def __init__(self):
        self.R = False # Clear
        self.C1 = False # Clock
        self.D1 = False # D Input
        self.Q1 = False # Q Output

    def clear_ic(self):
        self.Q1 = False

    def clock_high(self):
        self.Q1 = self.D1
```

```

def __repr__(self):
    return ("D Flip-Flop: " + "R: " + str(self.R) + " C1: " + str(self.C1)
+
    " D1: " + str(self.D1) + " Q1: " + str(self.Q1))

class IC74273(IC):
    """Model for 74ls273 Quad D Flip-Flop IC - 20-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic74273"

        self.logic_dict = {}
        self.count_int = 0
        self.conn_inc = 15
        self.offset = -5

        # Initialize 8 x D-flip flops
        self.d1 = DFlipFlop()
        self.d2 = DFlipFlop()
        self.d3 = DFlipFlop()
        self.d4 = DFlipFlop()
        self.d5 = DFlipFlop()
        self.d6 = DFlipFlop()
        self.d7 = DFlipFlop()
        self.d8 = DFlipFlop()
        self.ff_list = [self.d1, self.d2, self.d3, self.d4, self.d5, self.d6,
self.d7, self.d8]

        # Set initial logic states
        for i in range(1, 21): # 20 pin IC
            self.logic_dict['c' + str(i)] = False

        self.filename = Path(__file__).parent /
"../images/ics/74273_easy_100x155.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_image(self.filename)

```

```

    self.update_bbox()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def create_connectors(self):
    # Calculate position of connectors from current shape position and size
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w, h = x2 - x1, y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)
    conn_inc = self.conn_inc
    offset = self.offset

    # Note: Connector names correspond to pin-numbers, c1 = pin 1
    # Left side connectors
    self.conn_list.append(Connector(self.canvas, "c1", center.x - w / 2,
center.y - h / 2 +
                           offset + conn_inc * 1))
    self.conn_list.append(Connector(self.canvas, "c3", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 2))
    self.conn_list.append(Connector(self.canvas, "c4", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 3))
    self.conn_list.append(Connector(self.canvas, "c7", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 4))
    self.conn_list.append(Connector(self.canvas, "c8", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 5))
    self.conn_list.append(Connector(self.canvas, "c13", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 6))
    self.conn_list.append(Connector(self.canvas, "c14", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 7))
    self.conn_list.append(Connector(self.canvas, "c17", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 8))
    self.conn_list.append(Connector(self.canvas, "c18", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 9))
    self.conn_list.append(Connector(self.canvas, "c10", center.x - w / 2,
center.y - h / 2 + offset +
                           conn_inc * 10))

```



```

(i+1))

    # Update right side pins
    for j in range(0, 10):
        self.conn_list[j+10].x, self.conn_list[j+10].y = (center.x + w / 2,
center.y - h / 2 +
                                         offset + conn_inc
* (j+1))

    # Draw the connectors
    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def set_logic_level(self):
    for wire in self.wire_list:
        # print(wire.connector_obj.name)
        if wire.connector_obj.name == "c1": # CLR input
            self.logic_dict['c1'] = wire.wire_obj.state
            if not wire.wire_obj.state:
                for ff in self.ff_list:
                    ff.clear_ic()
            self.set_q_outputs()
        elif wire.connector_obj.name == "c11": # CLK input
            self.logic_dict['c9'] = wire.wire_obj.state
            if wire.wire_obj.state:
                for ff in self.ff_list:
                    ff.Q1 = ff.D1
            self.set_q_outputs()
        elif wire.connector_obj.name == "c2": # 1Q output
            wire.wire_obj.state = self.logic_dict['c2']
        elif wire.connector_obj.name == "c3": # 1D input
            self.d1.D1 = wire.wire_obj.state
            self.logic_dict['c3'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c4": # 2D input
            self.d2.D1 = wire.wire_obj.state
            self.logic_dict['c4'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c5": # 2Q Output
            wire.wire_obj.state = self.logic_dict['c5']
        elif wire.connector_obj.name == "c6": # 3Q Output
            wire.wire_obj.state = self.logic_dict['c6']
        elif wire.connector_obj.name == "c7": # 3D input
            self.d3.D1 = wire.wire_obj.state
            self.logic_dict['c7'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c8": # 4D input

```

```

        self.d4.D1 = wire.wire_obj.state
        self.logic_dict['c8'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c9": # 4Q output
        wire.wire_obj.state = self.logic_dict['c9']
    elif wire.connector_obj.name == "c10": # GND - input
        self.logic_dict['c10'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c12": # 5Q output
        wire.wire_obj.state = self.logic_dict['c12']
    elif wire.connector_obj.name == "c13": # 5D input
        self.d5.D1 = wire.wire_obj.state
        self.logic_dict['c13'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c14": # 6D input
        self.d6.D1 = wire.wire_obj.state
        self.logic_dict['c14'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c15": # 6Q output
        wire.wire_obj.state = self.logic_dict['c15']
    elif wire.connector_obj.name == "c16": # 7Q output
        wire.wire_obj.state = self.logic_dict['c16']
    elif wire.connector_obj.name == "c17": # 7D input
        self.d7.D1 = wire.wire_obj.state
        self.logic_dict['c17'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c18": # 8D input
        self.d8.D1 = wire.wire_obj.state
        self.logic_dict['c18'] = wire.wire_obj.state
    elif wire.connector_obj.name == "c19": # 8Q output
        wire.wire_obj.state = self.logic_dict['c19']
    elif wire.connector_obj.name == "c20": # VCC input
        self.logic_dict['c20'] = wire.wire_obj.state

def set_q_outputs(self):
    self.logic_dict['c2'] = self.d1.Q1 # 1Q
    self.logic_dict['c5'] = self.d2.Q1 # 2Q
    self.logic_dict['c6'] = self.d3.Q1 # 3Q
    self.logic_dict['c9'] = self.d4.Q1 # 4Q
    self.logic_dict['c12'] = self.d1.Q1 # 5Q
    self.logic_dict['c15'] = self.d2.Q1 # 6Q
    self.logic_dict['c16'] = self.d3.Q1 # 7Q
    self.logic_dict['c19'] = self.d4.Q1 # 8Q

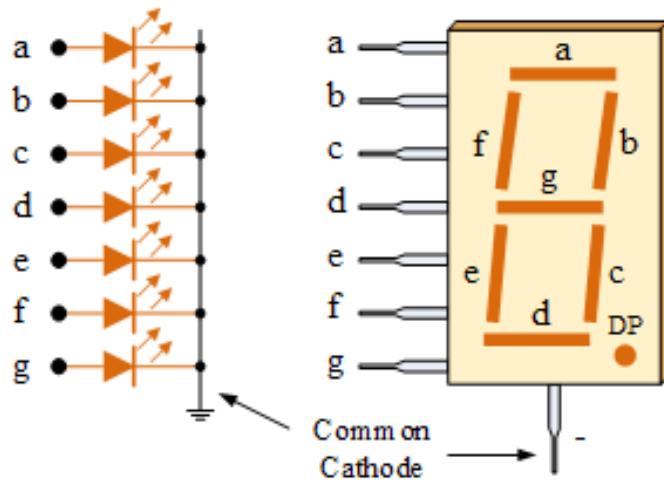
```

7-Segment Display Class

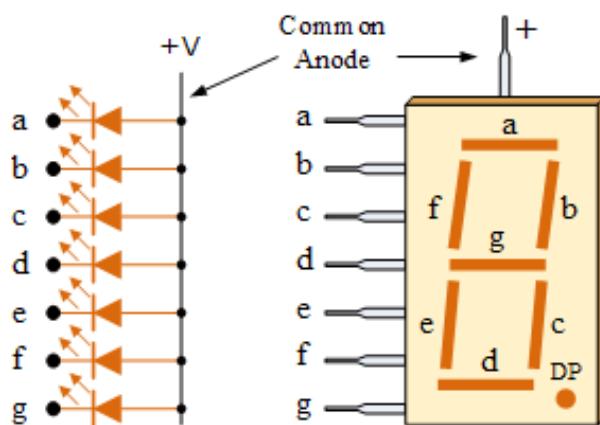
Objectives:

- Create a new 7-Segment display class
- Draw the display using shapes so that segment colors are programmable

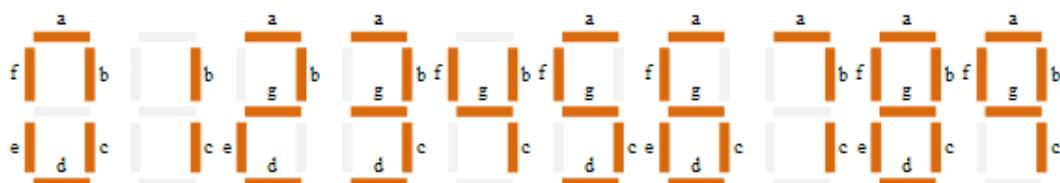
Common Cathode Configuration



Common Anode Configuration

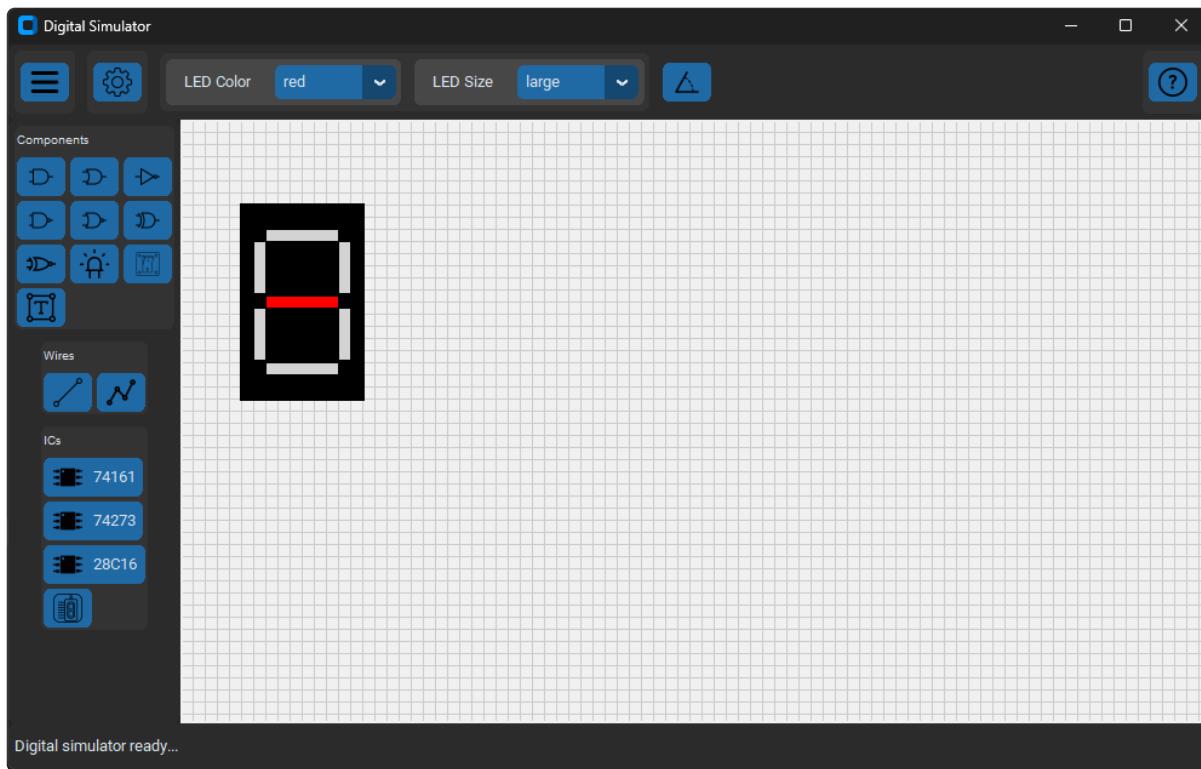


Segments needed for all Numbers



Truth Table

Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	x	x	x	x	x	x	
1		x	x				
2	x	x		x	x		x
3	x	x	x	x			x
4		x	x			x	x
5	x		x	x		x	x
6	x		x	x	x	x	x
7	x	x	x				
8	x	x	x	x	x	x	x
9	x	x	x			x	x



IC.Lib/ic_7_segment_display.py

```
from pathlib import Path

from IC.Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class Segment:
    def __init__(self, canvas, x1, y1, orientation):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.orientation = orientation

        if self.orientation == "H":
            self.w = 60
            self.h = 10
        elif self.orientation == "V":
            self.w = 10
            self.h = 43

        self.x2 = self.x1 + self.w
        self.y2 = self.y1 + self.h
```

```

        self.id = None
        self.state = False

        self.create_segment()

    def create_segment(self):
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2, fill="light gray")

    def update(self):
        self.update_position()
        self.update_color()

    def update_position(self):
        """Update the position when the object is moved"""
        self.canvas.coords(self.x1, self.y1, self.x2, self.y2)

    def update_color(self):
        if self.state:
            self.canvas.itemconfig(self.id, fill="red")
        else:
            self.canvas.itemconfig(self.id, fill="light gray")

class SevenSegment(IC):
    """Model for 7-Segment Display - 7-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "7_segment"

        # IC dimensions
        self.w = 100
        self.h = 160
        self.bbox = None

        # Initialize shape ids
        self.id = None
        self.seg_a, self.seg_b, self.seg_c, self.seg_d, self.seg_e, self.seg_f,
self.seg_g = (
            None, None, None, None, None, None, None)
        self.seg_list = []

        self.create_display()
        self.create_segments()
        self.update_bbox()
        self.create_selector()

```

```

def create_display(self):
    # Assume black background with red segments
    self.id = self.canvas.create_rectangle(self.x1 - self.w/2, self.y1 -
self.h/2,
                                            self.x1 + self.w/2, self.y1 +
self.h/2,
                                            fill="black", outline="black",
width=3)

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_segments()
    self.update_selector()

def update_position(self):
    """Update the position when the object is moved"""
    self.canvas.coords(self.x1 - self.w/2, self.y1 - self.h/2,
                       self.x1 + self.w/2, self.y1 + self.h/2)

def create_segments(self):
    # Horizontal segments
    self(seg_a = Segment(self.canvas, self.x1 - self.w/2 + 20, self.y1 -
self.h/2 + 20, "H"))
    self.seg_list.append(self.seg_a)
    self(seg_g = Segment(self.canvas, self.x1 - self.w/2 + 20, self.y1 - 5,
"H"))
    self.seg_list.append(self.seg_g)
    self.seg_g.state = True
    self(seg_d = Segment(self.canvas, self.x1 - self.w/2 + 20, self.y1 +
self.h/2 - 30, "H"))
    self.seg_list.append(self.seg_d)

    # Vertical segments
    self(seg_f = Segment(self.canvas, self.x1 - self.w/2 + 10, self.y1 -
self.h/2 + 30, "V"))
    self.seg_list.append(self.seg_f)
    self(seg_b = Segment(self.canvas, self.x1 + self.w/2 - 20, self.y1 -
self.h/2 + 30, "V"))
    self.seg_list.append(self.seg_b)
    self(seg_e = Segment(self.canvas, self.x1 - self.w/2 + 10, self.y1 + 5,
"V"))
    self.seg_list.append(self.seg_e)
    self(seg_c = Segment(self.canvas, self.x1 + self.w/2 - 20, self.y1 + 5,
"V"))
    self.seg_list.append(self.seg_c)

```

```

    self(seg_list.append(self.seg_c)

def update_segments(self):
    for s in self(seg_list:
        s.update()

# TODO: Add connectors and set_logic_level

```

UI_Lib/ic_button_frame.py

```

import customtkinter as ctk
from pathlib import Path
from PIL import Image

from IC_Lib import IC74161, IC74273, IC28C16, SevenSegment


class ICButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="ICs", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        ic_image = ctk.CTkImage(light_image=Image.open
            (Path(__file__).parent / "../icons/ic.png"),
            dark_image=Image.open
            (Path(__file__).parent / "../icons/ic.png"),
            size=(24, 24))

        ic_button = ctk.CTkButton(self, text="74161", image=ic_image, width=30,
                                command=self.create_ic_74161)
        ic_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)

        ic_button = ctk.CTkButton(self, text="74273", image=ic_image, width=30,
                                command=self.create_ic_74273)
        ic_button.grid(row=2, column=0, sticky=ctk.W, padx=2, pady=2)

```

```
ic_button = ctk.CTkButton(self, text="28C16", image=ic_image, width=30,
                           command=self.create_ic_28C16)
ic_button.grid(row=3, column=0, sticky=ctk.W, padx=2, pady=2)

seven_segment_image = ctk.CTkImage(light_image=Image.open # Added new
image
                                    (Path(__file__).parent / "../icons/7-segment-display.png"),
                                    dark_image=Image.open
                                    (Path(__file__).parent / "../icons/7-segment-display.png"),
                                    size=(24, 24))

seven_segment_button = ctk.CTkButton(self, text="",
                                     image=seven_segment_image, width=30, # Added new button
                                     command=self.create_seven_segment)
seven_segment_button.grid(row=4, column=0, sticky=ctk.W, padx=2,
                           pady=2) # Added new button

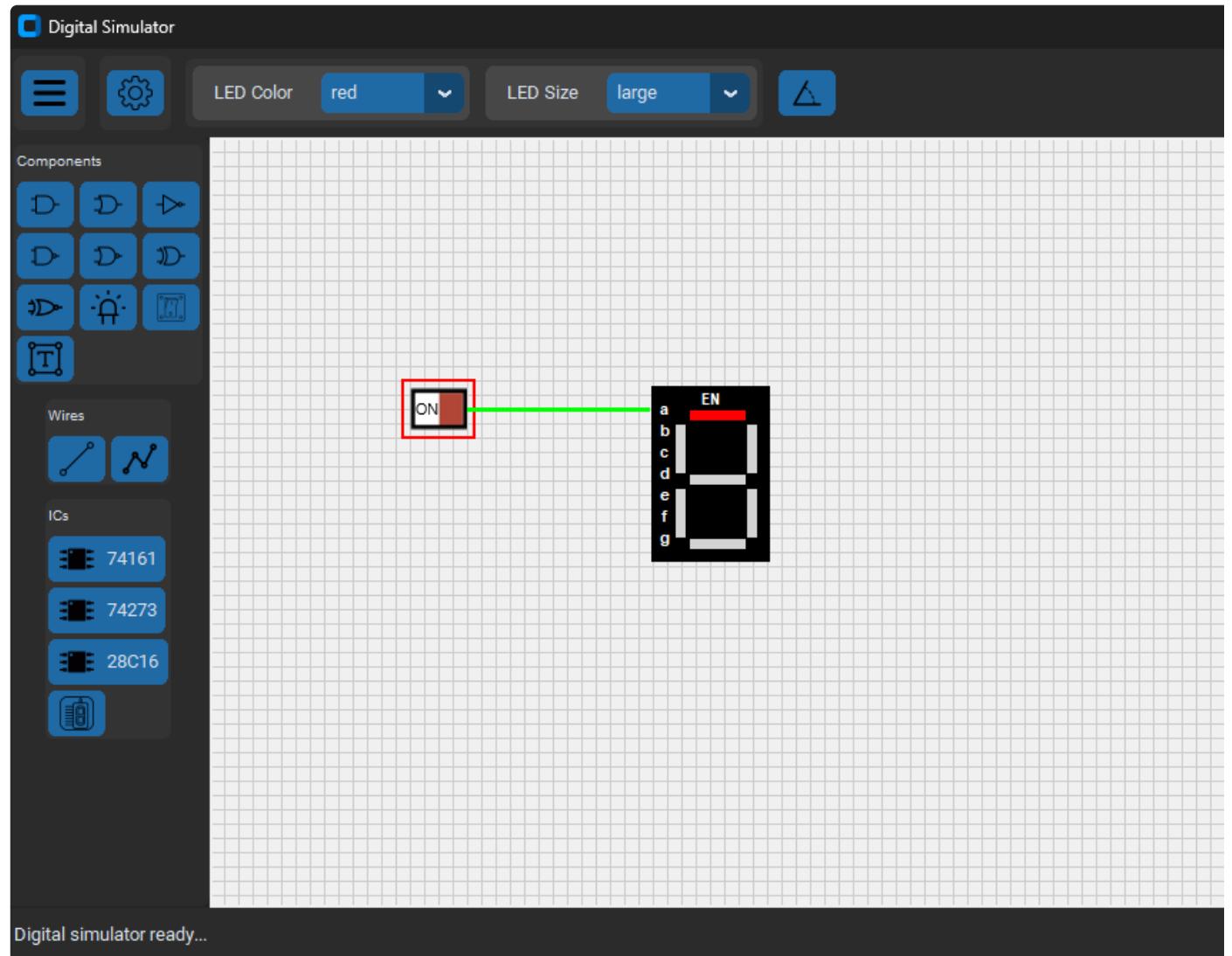
def create_ic_74161(self):
    ic = IC74161(self.canvas, 100, 100)
    self.canvas.comp_list.append(ic)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

def create_ic_74273(self):
    ic = IC74273(self.canvas, 105, 100)
    self.canvas.comp_list.append(ic)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

def create_ic_28C16(self):
    ic = IC28C16(self.canvas, 100, 150)
    self.canvas.comp_list.append(ic)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

def create_seven_segment(self): # Added new method
    ic = SevenSegment(self.canvas, 100, 150)
    self.canvas.comp_list.append(ic)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()
```

Connectors and Logic



IC.Lib/ic_7_segment_display.py

```
import customtkinter as ctk
from IC.Lib.ic import IC
from Helper.Lib.point import Point
from Wire.Lib.connector import Connector
from Comp.Lib import Text

class Segment:
    def __init__(self, canvas, x1, y1, orientation):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.orientation = orientation
```

```

        if self.orientation == "H":
            self.w = 40
            self.h = 8
        elif self.orientation == "V":
            self.w = 8
            self.h = 35

        self.x2 = self.x1 + self.w
        self.y2 = self.y1 + self.h
        self.id = None
        self.state = False

        self.create_segment()

    def create_segment(self):
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2, fill="light gray")

    def update(self):
        self.update_position()
        self.update_color()

    def update_position(self):
        self.x2 = self.x1 + self.w
        self.y2 = self.y1 + self.h
        """Update the position when the object is moved"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)

    def update_color(self):
        if self.state:
            self.canvas.itemconfig(self.id, fill="red")
        else:
            self.canvas.itemconfig(self.id, fill="light gray")

class SevenSegment(IC):
    """Model for 7-Segment Display - 7-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "7_segment"

        # IC dimensions
        self.w = 80
        self.h = 120
        self.bbox = None

```

```

        self.conn_inc = 15
        self.label_list = []
        self.logic_dict = {'a': False, 'b': False, 'c': False, 'd': False, 'e':
False, 'f': False, 'g': False,
                           'EN': False}

        # Initialize shape ids
        self.id = None
        self(seg_a, self(seg_b, self(seg_c, self(seg_d, self(seg_e, self(seg_f,
self(seg_g = (
                           None, None, None, None, None, None, None)
        self(seg_list = []

        self.create_display()
        self.create_segments()
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()
        self.create_labels()

    def create_display(self):
        # Assume black background with red segments
        self.id = self.canvas.create_rectangle(self.x1 - self.w/2, self.y1 -
self.h/2,
                                               self.x1 + self.w/2, self.y1 +
self.h/2,
                                               fill="black", outline="black",
width=3, tags='display')

    def update(self):
        self.set_logic_level()
        self.update_position()
        self.update_bbox()
        self.update_segments()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()
        self.update_labels()

    def update_position(self):
        """Update the position when the object is moved"""
        self.canvas.coords(self.id, self.x1 - self.w/2, self.y1 - self.h/2,
                           self.x1 + self.w/2, self.y1 + self.h/2)

```

```

def create_segments(self):
    # Horizontal segments
    self(seg_a = Segment(self.canvas, self.x1 - self.w/2 + 25, self.y1 -
self.h/2 + 15, "H"))
        self.seg_list.append(self.seg_a)
    self(seg_g = Segment(self.canvas, self.x1 - self.w/2 + 25, self.y1,
"H"))
        self.seg_list.append(self.seg_g)
    self(seg_d = Segment(self.canvas, self.x1 - self.w/2 + 25, self.y1 +
self.h/2 - 15, "H"))
        self.seg_list.append(self.seg_d)

    # Vertical segments
    self(seg_f = Segment(self.canvas, self.x1 - self.w/2 + 15, self.y1 -
self.h/2 + 25, "V"))
        self.seg_list.append(self.seg_f)
    self(seg_b = Segment(self.canvas, self.x1 + self.w/2 - 25, self.y1 -
self.h/2 + 25, "V"))
        self.seg_list.append(self.seg_b)
    self(seg_e = Segment(self.canvas, self.x1 - self.w/2 + 15, self.y1 +
10, "V"))
        self.seg_list.append(self.seg_e)
    self(seg_c = Segment(self.canvas, self.x1 + self.w/2 - 25, self.y1 +
10, "V"))
        self.seg_list.append(self.seg_c)

def update_segments(self):
    self.seg_a.x1, self.seg_a.y1 = self.x1 - self.w/2 + 25, self.y1 -
self.h/2 + 15
    self.seg_b.x1, self.seg_b.y1 = self.x1 + self.w/2 - 15, self.y1 -
self.h/2 + 25
    self.seg_c.x1, self.seg_c.y1 = self.x1 + self.w/2 - 15, self.y1 + 10
    self.seg_d.x1, self.seg_d.y1 = self.x1 - self.w/2 + 25, self.y1 +
self.h/2 - 15
    self.seg_e.x1, self.seg_e.y1 = self.x1 - self.w/2 + 15, self.y1 + 10
    self.seg_f.x1, self.seg_f.y1 = self.x1 - self.w/2 + 15, self.y1 -
self.h/2 + 25
    self.seg_g.x1, self.seg_g.y1 = self.x1 - self.w/2 + 25, self.y1
    for s in self.seg_list:
        s.update()

def create_connectors(self):
    # Calculate position of connectors from current shape position and size
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w, h = x2 - x1, y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

```

```

        conn_inc = self.conn_inc

        # Left side connectors
        self.conn_list.append(Connector(self.canvas, "a", center.x - w / 2,
center.y - conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "b", center.x - w / 2,
center.y - conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "c", center.x - w / 2,
center.y - conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "d", center.x - w / 2,
center.y))
        self.conn_list.append(Connector(self.canvas, "e", center.x - w / 2,
center.y + conn_inc * 1))
        self.conn_list.append(Connector(self.canvas, "f", center.x - w / 2,
center.y + conn_inc * 2))
        self.conn_list.append(Connector(self.canvas, "g", center.x - w / 2,
center.y + conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "EN", center.x, center.y -
h / 2))

    def update_connectors(self):
        # Recalculate position of connectors from current shape position and
size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc

        self.conn_list[0].x, self.conn_list[0].y = center.x - w / 2, center.y -
conn_inc * 3
        self.conn_list[1].x, self.conn_list[1].y = center.x - w / 2, center.y -
conn_inc * 2
        self.conn_list[2].x, self.conn_list[2].y = center.x - w / 2, center.y -
conn_inc * 1
        self.conn_list[3].x, self.conn_list[3].y = center.x - w / 2, center.y
        self.conn_list[4].x, self.conn_list[4].y = center.x - w / 2, center.y +
conn_inc * 1
        self.conn_list[5].x, self.conn_list[5].y = center.x - w / 2, center.y +
conn_inc * 2
        self.conn_list[6].x, self.conn_list[6].y = center.x - w / 2, center.y +
conn_inc * 3

        self.conn_list[7].x, self.conn_list[7].y = center.x, center.y - h / 2

        # Draw the connectors
        for c in self.conn_list:

```

```

c.update()

self.move_connected_wires()

def create_labels(self):
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w, h = x2 - x1, y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)
    conn_inc = self.conn_inc
    pos = 10

    self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y - conn_inc * 3, text='a', fill="white"))
    self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y - conn_inc * 2, text='b', fill="white"))
    self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y - conn_inc * 1, text='c', fill="white"))
    self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y, text='d', fill="white"))
    self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y + conn_inc * 1, text='e', fill="white"))
    self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y + conn_inc * 2, text='f', fill="white"))
    self.label_list.append(Text(self.canvas, center.x - w / 2 + pos,
center.y + conn_inc * 3, text='g', fill="white"))
    self.label_list.append(Text(self.canvas, center.x, center.y - h / 2 +
pos, text='EN', fill="white"))

def update_labels(self):
    # Recalculate position of connectors from current shape position and
    size
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w, h = x2 - x1, y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)
    conn_inc = self.conn_inc
    pos = 10

    self.label_list[0].x1, self.label_list[0].y1 = center.x - w / 2 + pos,
    center.y - conn_inc * 3
    self.label_list[1].x1, self.label_list[1].y1 = center.x - w / 2 + pos,
    center.y - conn_inc * 2
    self.label_list[2].x1, self.label_list[2].y1 = center.x - w / 2 + pos,
    center.y - conn_inc * 1
    self.label_list[3].x1, self.label_list[3].y1 = center.x - w / 2 + pos,
    center.y
    self.label_list[4].x1, self.label_list[4].y1 = center.x - w / 2 + pos,

```

```

center.y + conn_inc * 1
    self.label_list[5].x1, self.label_list[5].y1 = center.x - w / 2 + pos,
center.y + conn_inc * 2
    self.label_list[6].x1, self.label_list[6].y1 = center.x - w / 2 + pos,
center.y + conn_inc * 3
    self.label_list[7].x1, self.label_list[7].y1 = center.x, center.y - h /
2 + pos

    for label in self.label_list:
        self.canvas.coords(label.id, label.x1, label.y1)

def set_logic_level(self):
    for wire in self.wire_list:
        if wire.connector_obj.name == "a":
            self(seg_a.state = wire.wire_obj.state
        elif wire.connector_obj.name == "b":
            self(seg_b.state = wire.wire_obj.state
        elif wire.connector_obj.name == "c":
            self(seg_c.state = wire.wire_obj.state
        elif wire.connector_obj.name == "d":
            self(seg_d.state = wire.wire_obj.state
        elif wire.connector_obj.name == "e":
            self(seg_e.state = wire.wire_obj.state
        elif wire.connector_obj.name == "f":
            self(seg_f.state = wire.wire_obj.state
        elif wire.connector_obj.name == "g":
            self(seg_g.state = wire.wire_obj.state
        elif wire.connector_obj.name == "EN":
            pass

```

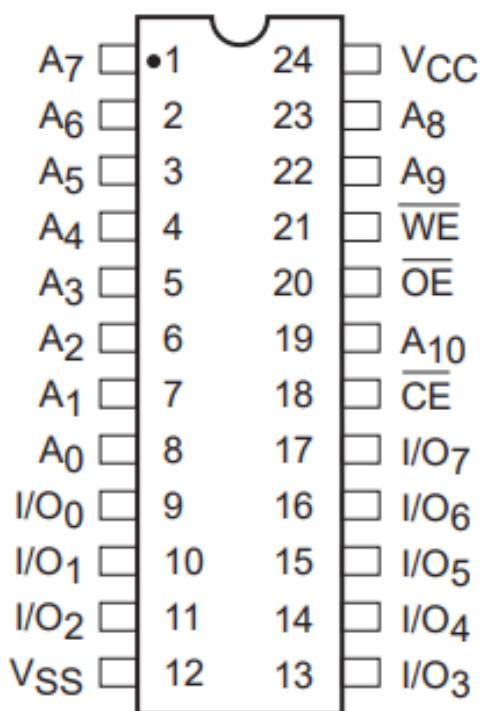
28C16 2K x 8-Bit EEPROM Class

Objectives:

- Create a 28C16 EEPROM class
- Create a "program" for converting binary input to decimal output for 7-segment display
- Program to support 4-bit conversion from 0x00 - 0xFF hex to 0 - 15 dec
- Binary input on address pins A0 to A10 - 11-bit address = 2048 addresses

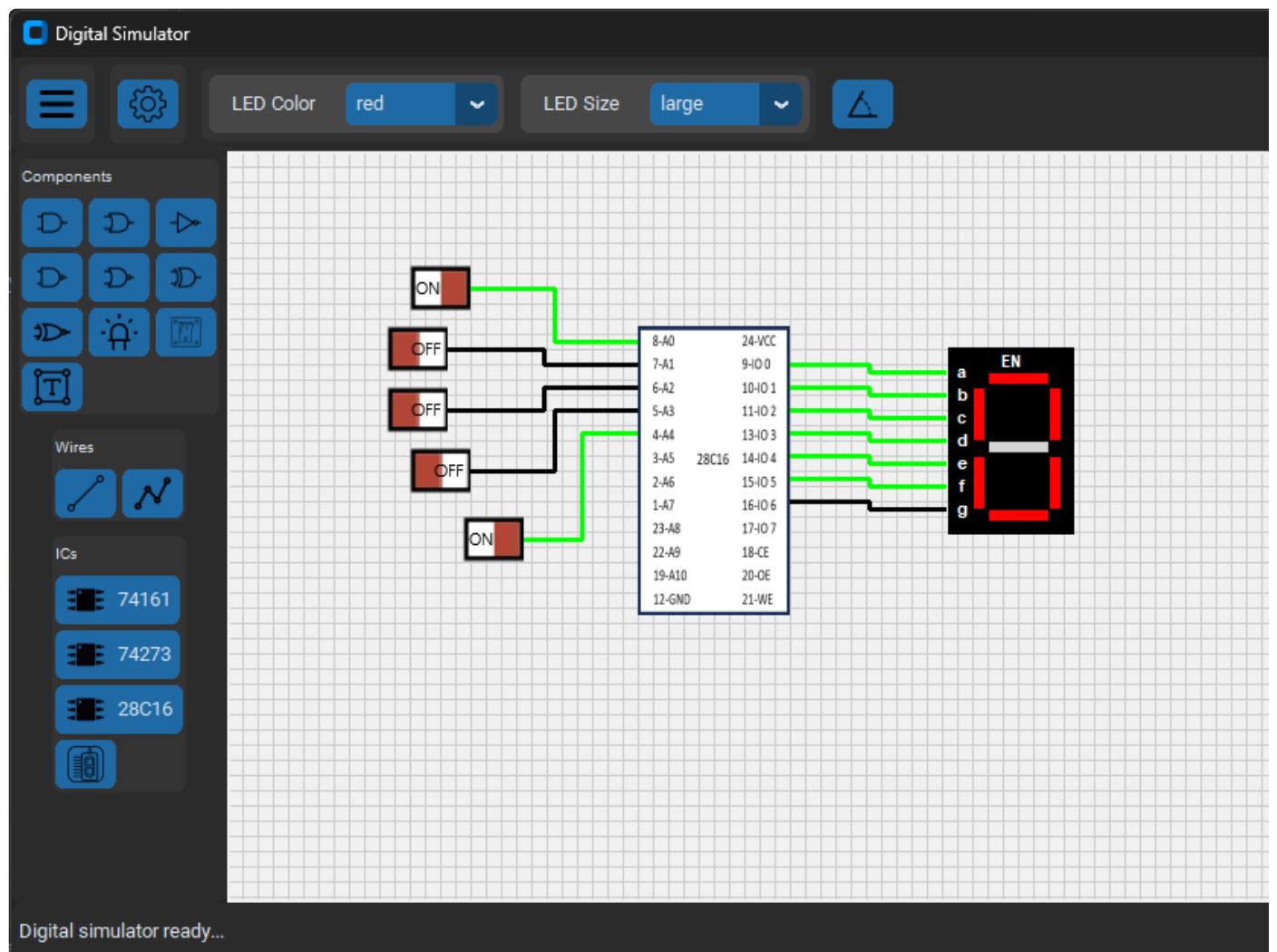
[28C16 Data Sheet](#)

Device pin-out



Device pin functions

Pin Name	Function
A ₀ -A ₁₀	Address Inputs
I/O ₀ -I/O ₇	Data Inputs/Outputs
CE	Chip Enable
OE	Output Enable
WE	Write Enable
V _{CC}	5V Supply
V _{SS}	Ground
NC	No Connect



IC_Lib/ic_28C16_eeprom.py

```
from pathlib import Path

from IC_Lib.ic import IC
from Helper_Lib.point import Point
from Wire_Lib.connector import Connector


class IC28C16(IC):
    """Model for 28C16 2K x 8-Bit EEPROM - 24-pin package"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "ic28C16"

        self.logic_dict = {}
        self.conn_inc = 15
        self.offset = -5
```

```

# Set initial logic states
for i in range(1, 25): # 24 pin IC
    self.logic_dict['c' + str(i)] = False

    self.zero, self.one, self.two, self.three, self.four, self.five,
self.six, self.seven, self.eight, self.nine = (
        None, None, None, None, None, None, None, None, None, None)
    self.set_display_data()

    self.filename = Path(__file__).parent /
"../images/ics/28C16_easy_100x190.png"
    self.create_ic()

# Define program for Display 0-1
self.program1 = {
    '0000': self.zero,
    '0001': self.zero,
    '0010': self.zero,
    '0011': self.zero,
    '0100': self.zero,
    '0101': self.zero,
    '0110': self.zero,
    '0111': self.zero,
    '1000': self.zero,
    '1001': self.zero,
    '1010': self.one,
    '1011': self.one,
    '1100': self.one,
    '1101': self.one,
    '1110': self.one,
    '1111': self.one
}

# Define program for Display 0-9
self.program2 = {
    '0000': self.zero,
    '0001': self.one,
    '0010': self.two,
    '0011': self.three,
    '0100': self.four,
    '0101': self.five,
    '0110': self.six,
    '0111': self.seven,
    '1000': self.eight,
    '1001': self.nine,
    '1010': self.zero,
}

```

```

        '1011': self.one,
        '1100': self.two,
        '1101': self.three,
        '1110': self.four,
        '1111': self.five
    }

def set_display_data(self):
    # Set output for a, b, c, d, e, f, g on the display
    self.zero = [True, True, True, True, True, True, False]
    self.one = [False, True, True, False, False, False, False]
    self.two = [True, True, False, True, True, False, True]
    self.three = [True, True, True, True, False, False, True]
    self.four = [False, True, True, False, False, True, True]
    self.five = [True, False, True, True, False, True, True]
    self.six = [True, False, True, True, True, True, True]
    self.seven = [True, True, True, False, False, False, False]
    self.eight = [True, True, True, True, True, True, True]
    self.nine = [True, True, True, False, False, True, True]

def create_ic(self):
    self.create_image(self.filename)
    self.update_bbox()
    self.create_selector()
    self.create_connectors()
    self.set_connector_visibility()

def update(self):
    self.set_logic_level()
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def create_connectors(self):
    # Calculate position of connectors from current shape position and size
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w, h = x2 - x1, y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)
    conn_inc = self.conn_inc
    offset = self.offset

    # Note: Connector names correspond to pin-numbers, c1 = pin 1
    # Left side connectors

```

```

        self.conn_list.append(Connector(self.canvas, "c8", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 1))
    self.conn_list.append(Connector(self.canvas, "c7", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 2))
    self.conn_list.append(Connector(self.canvas, "c6", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 3))
    self.conn_list.append(Connector(self.canvas, "c5", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 4))
    self.conn_list.append(Connector(self.canvas, "c4", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 5))
    self.conn_list.append(Connector(self.canvas, "c3", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 6))
    self.conn_list.append(Connector(self.canvas, "c2", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 7))
    self.conn_list.append(Connector(self.canvas, "c1", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 8))
    self.conn_list.append(Connector(self.canvas, "c23", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 9))
    self.conn_list.append(Connector(self.canvas, "c22", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 10))
    self.conn_list.append(Connector(self.canvas, "c19", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 11))
    self.conn_list.append(Connector(self.canvas, "c12", center.x - w / 2,
center.y - h / 2 + offset +
                                conn_inc * 12))

# Right side connectors
    self.conn_list.append(Connector(self.canvas, "c24", center.x + w / 2,
center.y - h/2 + offset +
                                conn_inc * 1))
    self.conn_list.append(Connector(self.canvas, "c9", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 2))
    self.conn_list.append(Connector(self.canvas, "c10", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 3))

```

```

                conn_inc * 3))
        self.conn_list.append(Connector(self.canvas, "c11", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 4))
        self.conn_list.append(Connector(self.canvas, "c13", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 5))
        self.conn_list.append(Connector(self.canvas, "c14", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 6))
        self.conn_list.append(Connector(self.canvas, "c15", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 7))
        self.conn_list.append(Connector(self.canvas, "c16", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 8))
        self.conn_list.append(Connector(self.canvas, "c17", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 9))
        self.conn_list.append(Connector(self.canvas, "c18", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 10))
        self.conn_list.append(Connector(self.canvas, "c20", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 11))
        self.conn_list.append(Connector(self.canvas, "c21", center.x + w / 2,
center.y - h / 2 + offset +
                                conn_inc * 12))

    def update_connectors(self):
        # Recalculate position of connectors from current shape position and
        size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)
        conn_inc = self.conn_inc
        offset = self.offset

        # Update left side pins
        for i in range(0, 12):
            self.conn_list[i].x, self.conn_list[i].y = (center.x - w / 2,
center.y - h / 2 +
                                offset + conn_inc *
(i+1))

        # Update right side pins

```

```

        for j in range(0, 12):
            self.conn_list[j+12].x, self.conn_list[j+12].y = (center.x + w / 2,
center.y - h / 2 +
                                         offset + conn_inc
* (j+1))

        # Draw the connectors
        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

def set_logic_level(self):
    for wire in self.wire_list:
        if wire.connector_obj.name == "c1":
            self.logic_dict['c1'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c2":
            self.logic_dict['c2'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c3":
            self.logic_dict['c3'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c4":
            self.logic_dict['c4'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c5":
            self.logic_dict['c5'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c6":
            self.logic_dict['c6'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c7":
            self.logic_dict['c7'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c8":
            self.logic_dict['c8'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c9": # I/O 0
            wire.wire_obj.state = self.logic_dict['c9']
        elif wire.connector_obj.name == "c10": # I/O 1
            wire.wire_obj.state = self.logic_dict['c10']
        elif wire.connector_obj.name == "c11": # I/O 2
            wire.wire_obj.state = self.logic_dict['c11']
        elif wire.connector_obj.name == "c12": # VSS = GND
            self.logic_dict['c12'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c13": # I/O 3
            wire.wire_obj.state = self.logic_dict['c13']
        elif wire.connector_obj.name == "c14": # I/O 4
            wire.wire_obj.state = self.logic_dict['c14']
        elif wire.connector_obj.name == "c15": # I/O 5
            wire.wire_obj.state = self.logic_dict['c15']
        elif wire.connector_obj.name == "c16": # I/O 6
            wire.wire_obj.state = self.logic_dict['c16']

```

```

        elif wire.connector_obj.name == "c17": # I/O 7
            wire.wire_obj.state = self.logic_dict['c17']
        elif wire.connector_obj.name == "c18": # !CE
            self.logic_dict['c18'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c19": # A10
            self.logic_dict['c19'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c20": # !OE
            self.logic_dict['c20'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c21": # !WE
            self.logic_dict['c21'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c22": # A9
            self.logic_dict['c22'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c23": # A8
            self.logic_dict['c23'] = wire.wire_obj.state
        elif wire.connector_obj.name == "c24": # VCC
            self.logic_dict['c24'] = wire.wire_obj.state

    key, ps = self.convert_address_to_key()
    if ps is True: # Select program #1
        io = self.program1[key]
    else: # Select program #2
        io = self.program2[key]

    self.set_io_output(io)

def convert_address_to_key(self):
    a0 = self.logic_dict['c8'] # Counter bit 0
    a1 = self.logic_dict['c7'] # Counter bit 1
    a2 = self.logic_dict['c6'] # Counter bit 2
    a3 = self.logic_dict['c5'] # Counter bit 3

    ps = self.logic_dict['c4'] # EEPROM Program Select

    result = lambda s: '1' if s is True else '0'
    k0 = result(a0)
    k1 = result(a1)
    k2 = result(a2)
    k3 = result(a3)
    key = k3 + k2 + k1 + k0
    return key, ps

def set_io_output(self, io):
    self.logic_dict['c9'] = io[0] # i/o 0 = a
    self.logic_dict['c10'] = io[1] # i/o 1 = b
    self.logic_dict['c11'] = io[2] # i/o 2 = c
    self.logic_dict['c13'] = io[3] # i/o 3 = d

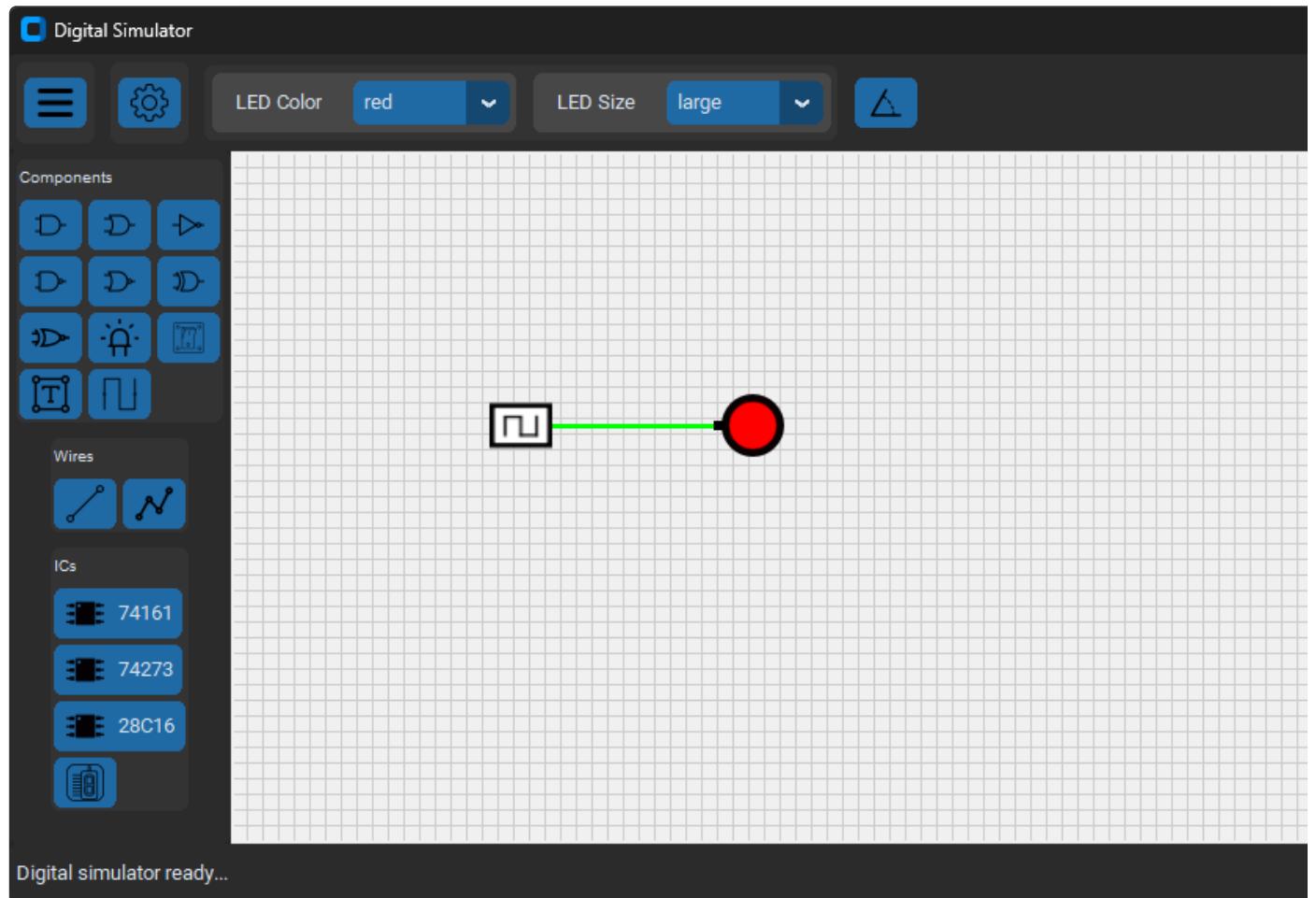
```

```
self.logic_dict['c14'] = io[4] # i/o 4 = e  
self.logic_dict['c15'] = io[5] # i/o 5 = f  
self.logic_dict['c16'] = io[6] # i/o 6 = g  
self.logic_dict['c17'] = False # Not Used
```

Clock Class

Objectives:

- Create a new clock class
- Create the logic on a separate thread to improve performance



Comp_Lib/clock.py

```
import threading  
import time
```

```

from Comp_Lib.component import Comp
from Wire_Lib.connector import Connector
from Helper_Lib.point import Point


class Clock(Comp):
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.type = "clock"
        self.out_state = False # OFF state

        self.filename =
"D:/EETools/DigitalSimulator/images/switch/clock_40x30.png"

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

        # Create 1 connector
        self.out1_id = None
        self.create_connectors()
        self.set_connector_visibility()

        # Start the clock
        self.thread = threading.Thread(target=self.toggle_clock)
        self.thread.start()

    def __del__(self):
        print('Clock destructor called')
        self.thread.join()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self): # Added new method
        """Create connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

```

```

        # Define 1 connector: out1
        self.out1_id = Connector(self.canvas, "out", center.x + w / 2,
center.y)
        self.conn_list = [self.out1_id]
        self.set_connector_visibility()

    def update_connectors(self): # Added new method
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Update connector position based on angle
        if self.angle == 0:
            self.out1_id.x, self.out1_id.y = center.x + w / 2, center.y
        elif self.angle == 90:
            self.out1_id.x, self.out1_id.y = center.x, center.y - h / 2
        elif self.angle == 180:
            self.out1_id.x, self.out1_id.y = center.x - w / 2, center.y
        elif self.angle == 270:
            self.out1_id.x, self.out1_id.y = center.x, center.y + h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def toggle_clock(self):
        # set the time
        time.sleep(1)

        # Toggle state
        self.out_state = not self.out_state

        if self.wire_list:
            self.wire_list[0].wire_obj.state = self.out_state
            self.canvas.redraw_no_grid()

        self.toggle_clock()

```

```
def redraw(self):
    self.delete('grid_line')
    self.grid.draw()
    self.tag_lower("grid_line")
    for c in self.comp_list:
        c.update()

def redraw_no_grid(self): # Added new method
    for c in self.comp_list:
        c.update()
```

UI_Lib/comp_button_frame.py

```
("clock", "../icons/clock.png")] # Added icon for
clock

self.init_frame_widgets()

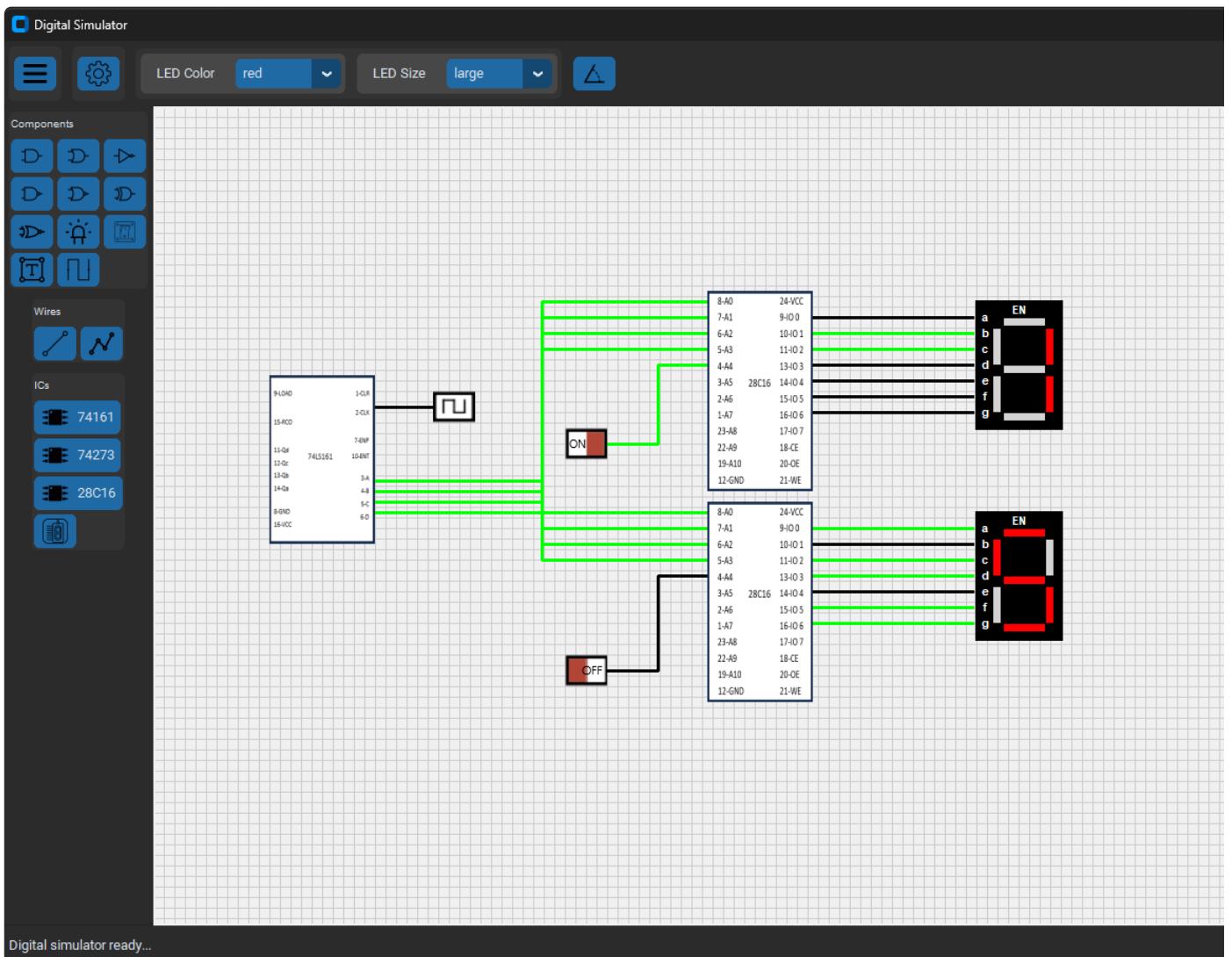
. . .

elif name == "text":
    gate = Text(self.canvas, 100, 100)
elif name == "clock": # Added clock class instantiation here
    gate = Clock(self.canvas, 100, 100)
self.canvas.comp_list.append(gate)
self.canvas.redraw()
self.canvas.mouse.move_mouse_bind_events()
```

Counter Circuit Simulation

Objective:

- Create a counter circuit
- 74161 Counter is driven by a Clock
- Add two 28C16 EEPROM
- Add two 7-segment displays
- Verify that the circuit counts from 0 to 15 and displays the count on the output displays
- Save and load the circuit from the file menu



Summary

This concludes the Digital Circuit Simulator advanced project development. The counter circuit simulation is pretty cool and we have developed all the modules to make it work. Python is definitely capable of complex circuit simulation. We will proceed to the next advanced project and my personal favorite - Microwave Circuit Simulation.

Chapter 7 - Microwave Circuit Simulator

Design, Features, and Specifications

RF & Microwave circuit design is a specialized field of electrical engineering for frequencies from 300 MHz to 100 GHz. This advanced project will address the design and development of a RF/Microwave Circuit Simulator. Note that analysis for high frequency circuits uses [S-Parameters](#) rather than Voltage and Current.

Program Design & Architecture:

- ✓ Modify the Digital Simulator to create the Microwave Simulator
- ✓ Object-Oriented Programming (OOP) - class abstraction, inheritance, & polymorphism
- ✓ KISS - Keep it simple, silly
- ✓ DRY - Don't repeat yourself
- ✓ SOC - Separation of concerns
- ✓ Python Modules & Packages
- ✓ User-interface
 - ✓ TopFrame class
 - ✓ File menu frame
 - ✓ Settings menu frame
 - ✓ Rotation button
 - ✓ Help menu frame
 - ✓ Left Frame Class with Circuit Component Menus
 - ✓ Canvas class
 - ✓ Mouse class
- ✓ Circuit components
 - ✓ Lumped Components
 - ✓ Resistor
 - ✓ Capacitor
 - ✓ Inductor
 - ✓ Ideal Components
 - ✓ Input Port
 - ✓ Output Port
 - ✓ Wires

- ✓ Straight Wire Class
- ✓ Segment Wire Class
- ✓ Elbow Wire Class
- ✓ Grid class
- ✓ Analysis
 - ✓ Linear S-Parameter Analysis
 - ✓ Raw data display
 - ✓ Line graphs
 - ✓ Smith chart

Key Technologies Needed:

- ✓ Microwave analysis library - Scikit-RF

Project Setup

Language: Python 3.11

IDE: PyCharm 2023.2.1 (Community Edition)

Project directory: D:/EETools/MicrowaveSimulator

Graphics library: CustomTkinter (<https://customtkinter.tomschimansky.com/>)

External libraries:

- ✓ pip install customtkinter
- ✓ python.exe -m pip install --upgrade pip
- ✓ pip install ctkcolorpicker
- ✓ pip install tkinter-tooltip
- ✓ pip install pyInstaller - Create .exe file
- ✓ pip install jsonpickle
- ✓ pip install matplotlib
- ✓ Add images and icons directories to the project.

RF/Microwave Analysis Library

- [scikit-rf](#)
- ✓ pip install scikit-rf

Scikit-RF Evaluation

- [Scikit-RF Home Page](#)
- [Scikit-RF Documentation](#)
- [Github Repository](#)

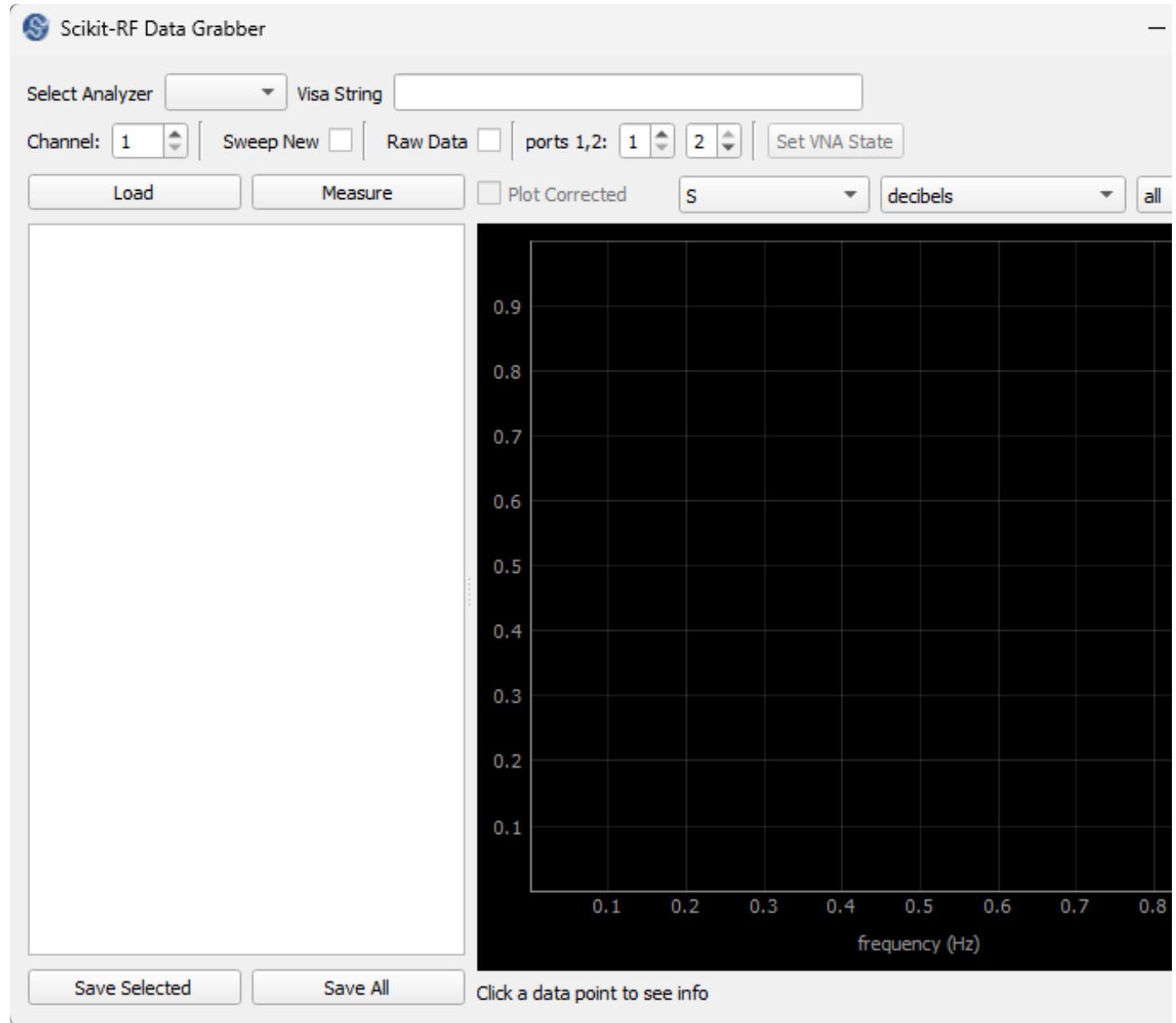
Features

- Microwave Network Operations:
 - Read/Write touchstone (.s2p, s?p) files
 - Arithmetic operations on scattering parameters
 - Cascade/De-embed 2-port networks
 - Frequency and port slicing and concatenation.
 - Connect n-port networks
 - s/z/y/abcd/t - parameter conversion
- Sets of Networks:
 - Statistical properties of NetworkSets
 - Methods to sort and visualize set behavior
- GUI support through [qtapps](#)
 - modular, re-useable apps
 - supports data retrieval, plotting, calibration and more.
- Plotting abilities:
 - Rectangular Plots (dB, mag, Phase, group delay)
 - Smith Chart
 - Automated Uncertainty bounds
- Offline Calibration:
 - One-Port: SOL, Least Squares, SDDL
 - Two-Port: TRL, Multiline TRL, SOLT, Unknown Thru, 8/16-Term
 - Partial : Enhanced Response, One-Port Two-Path
- Virtual Instruments (completeness varies by model)
 - VNAs: PNA, PNAX, ZVA, HP8510, HP8720
 - SA: HP8500
 - Others: ESP300
- Transmission Line Physics:
 - Coax, CPW, Freespace, Rectangular Waveguide, Distributed Circuit

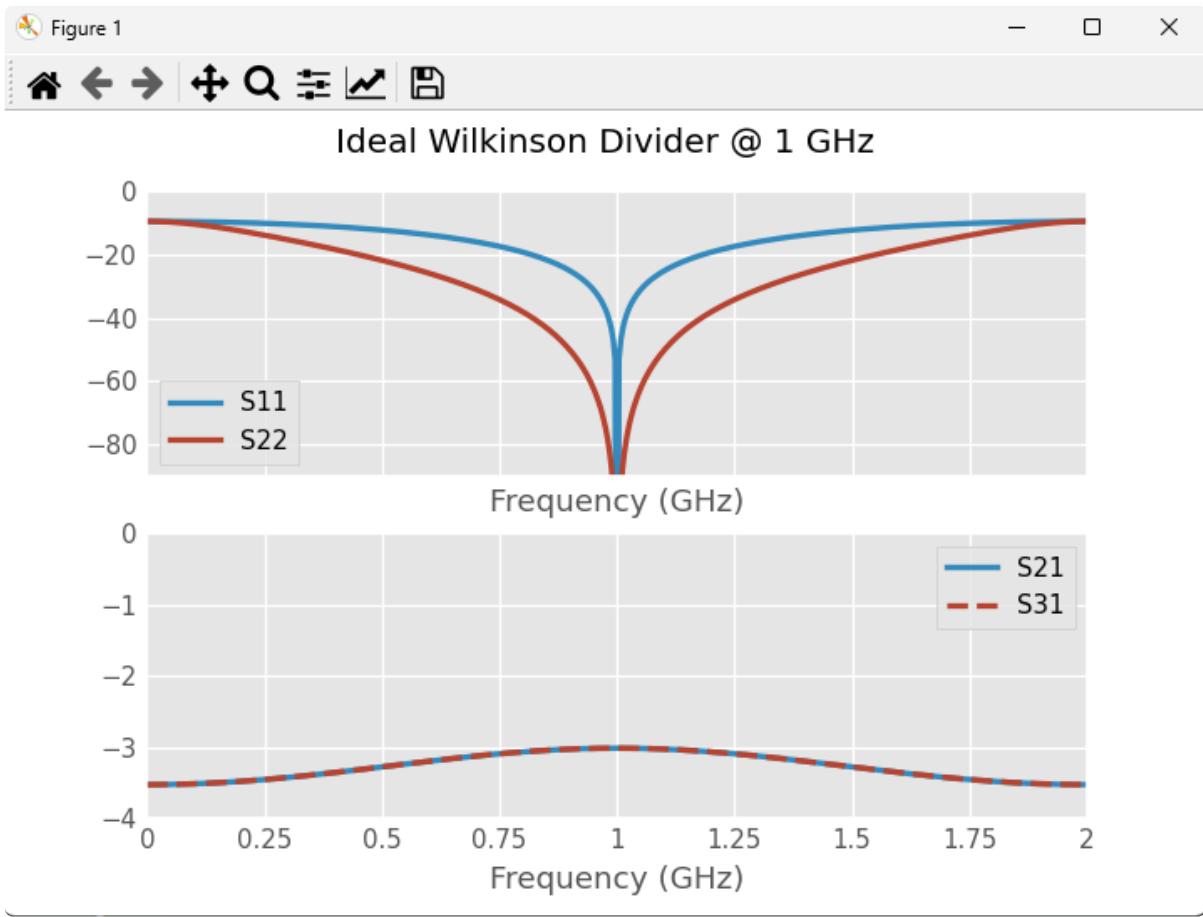
Media Classes

<u>DefinedGammaZ0</u> ([frequency, z0_port, z0, Z0, ...])	A media directly defined by its propagation constant and characteristic impedance.
<u>DistributedCircuit</u> ([frequency, z0_port, ...])	A transmission line mode defined in terms of distributed impedance and admittance values.
<u>RectangularWaveguide</u> ([frequency, z0_port, ...])	A single mode of a homogeneously filled rectangular waveguide.
<u>CircularWaveguide</u> ([frequency, z0_port, ...])	A single mode of a homogeneously filled Circular Waveguide
<u>CPW</u> ([frequency, z0_port, z0_override, z0, ...])	Coplanar waveguide.
<u>Coaxial</u> ([frequency, z0_port, z0_override, ...])	A coaxial transmission line defined in terms of its inner/outer diameters and permittivity.
<u>MLine</u> ([frequency, z0_port, z0_override, z0, ...])	A microstripline transmission line defined in terms of width, thickness and height on a given relative permittivity substrate.
<u>Freespace</u> ([frequency, z0_port, z0_override, ...])	A plane-wave (TEM Mode) in Freespace.
<u>DefinedAEpTandZ0</u> ([frequency, A, f_A, ep_r, ...])	Transmission line medium defined by A, Ep, Tand and Z0.

Example of a Scikit-rf GUI application



Wilkinson Power Divider Example



Sandbox/wilkinson_example.py

```

import numpy as np
import matplotlib.pyplot as plt
import skrf as rf

rf.stylely()

# frequency band
freq = rf.Frequency(start=0, stop=2, npoints=501, unit='GHz')

# characteristic impedance of the ports
z0_ports = 50

# resistor
R = 100
line_resistor = rf.media.DefinedGammaZ0(frequency=freq, z0=R)
resistor = line_resistor.resistor(R, name='resistor')

# branches
z0_branches = np.sqrt(2)*z0_ports
beta = freq.w/rf.c

```

```

line_branches = rf.media.DefinedGammaZ0(frequency=freq, z0=z0_branches,
gamma=0+beta*1j)

d = line_branches.theta_2_d(90, deg=True) # @ 90°(lambda/4)@ 1 GHz is ~ 75 mm
branch1 = line_branches.line(d, unit='m', name='branch1')
branch2 = line_branches.line(d, unit='m', name='branch2')

# ports
port1 = rf.Circuit.Port(freq, name='port1', z0=50)
port2 = rf.Circuit.Port(freq, name='port2', z0=50)
port3 = rf.Circuit.Port(freq, name='port3', z0=50)

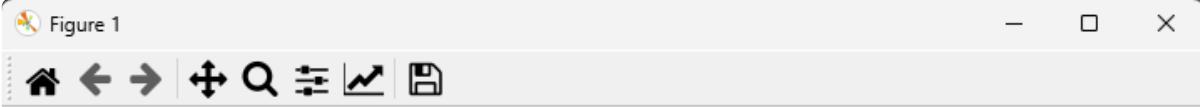
# Connection setup
# Note that the order of appearance of the port in the setup is important
connections = [
    [(port1, 0), (branch1, 0), (branch2, 0)],
    [(port2, 0), (branch1, 1), (resistor, 0)],
    [(port3, 0), (branch2, 1), (resistor, 1)]
]

# Building the circuit
C = rf.Circuit(connections)

fig, (ax1,ax2) = plt.subplots(2, 1, sharex=True)
C.network.plot_s_db(ax=ax1, m=0, n=0, lw=2) # S11
C.network.plot_s_db(ax=ax1, m=1, n=1, lw=2) # S22
ax1.set_ylim(-90, 0)
C.network.plot_s_db(ax=ax2, m=1, n=0, lw=2) # S21
C.network.plot_s_db(ax=ax2, m=2, n=0, ls='--', lw=2) # S31
ax2.set_ylim(-4, 0)
fig.suptitle('Ideal Wilkinson Divider @ 1 GHz')
plt.show()

```

Lumped Element Example



Sandbox/lumped_example.py

```

import numpy as np # for np.allclose() to check that S-params are similar
import skrf as rf
import matplotlib.pyplot as plt
rf.stylely()

# reference LC circuit made in Designer
LC_designer = rf.Network('data/designer_capacitor_30_80MHz_simple.s2p')

# scikit-rf: manually connecting networks
line = rf.media.DefinedGammaZ0(frequency=LC_designer.frequency, z0=50)
LC_manual = line.inductor(24e-9) ** line.capacitor(70e-12)

# scikit-rf: using Circuit builder
port1 = rf.Circuit.Port(frequency=LC_designer.frequency, name='port1', z0=50)
port2 = rf.Circuit.Port(frequency=LC_designer.frequency, name='port2', z0=50)
cap = rf.Circuit.SeriesImpedance(frequency=LC_designer.frequency, name='cap',
z0=50,
                           Z=1/(1j*LC_designer.frequency.w*70e-12))
ind = rf.Circuit.SeriesImpedance(frequency=LC_designer.frequency, name='ind',
z0=50,
                           Z=1/(1j*LC_designer.frequency.w*24e-9))
    
```

```

Z=1j*LC_designer.frequency.w*24e-9)

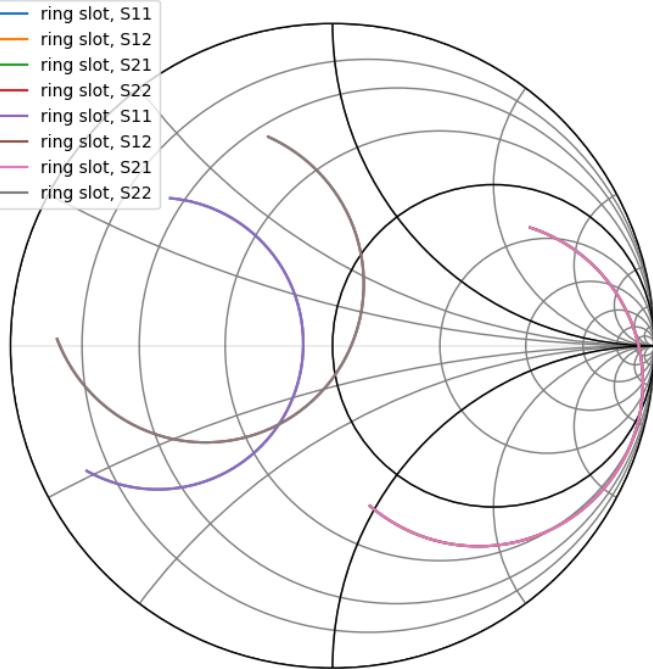
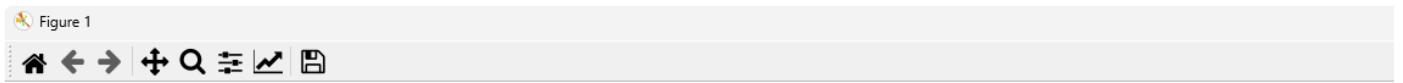
# NB: it is also possible to create 2-port lumped elements like:
# line = rf.media.DefinedGammaZ0(frequency=LC_designer.frequency, z0=50)
# cap = line.capacitor(70e-12, name='cap')
# ind = line.inductor(24e-9, name='ind')

connections = [
    [(port1, 0), (cap, 0)],
    [(cap, 1), (ind, 0)],
    [(ind, 1), (port2, 0)]
]
circuit = rf.Circuit(connections)
LC_from_circuit = circuit.network

LC_from_circuit.plot_s_db(m=0, n=0, lw=2)
LC_from_circuit.plot_s_db(m=1, n=0, lw=2)
LC_from_circuit.plot_s_db(m=0, n=0, lw=2, ls='-.')
LC_from_circuit.plot_s_db(m=1, n=0, lw=2, ls='-.')
plt.show()

```

Smith Chart Example



Lumped.smith_chart_example.py

```
import skrf as rf
import numpy as np
import matplotlib.pyplot as plt

network = rf.Network('data/ring_slot.s2p')
network.plot_s_smith()

frequencies = np.linspace(0, 200e9, 201)
network.plot_s_smith()
plt.show()
```

Main User Interface

The initial user interface design should look familiar by now. It is the same UI layout used for the digital simulator.

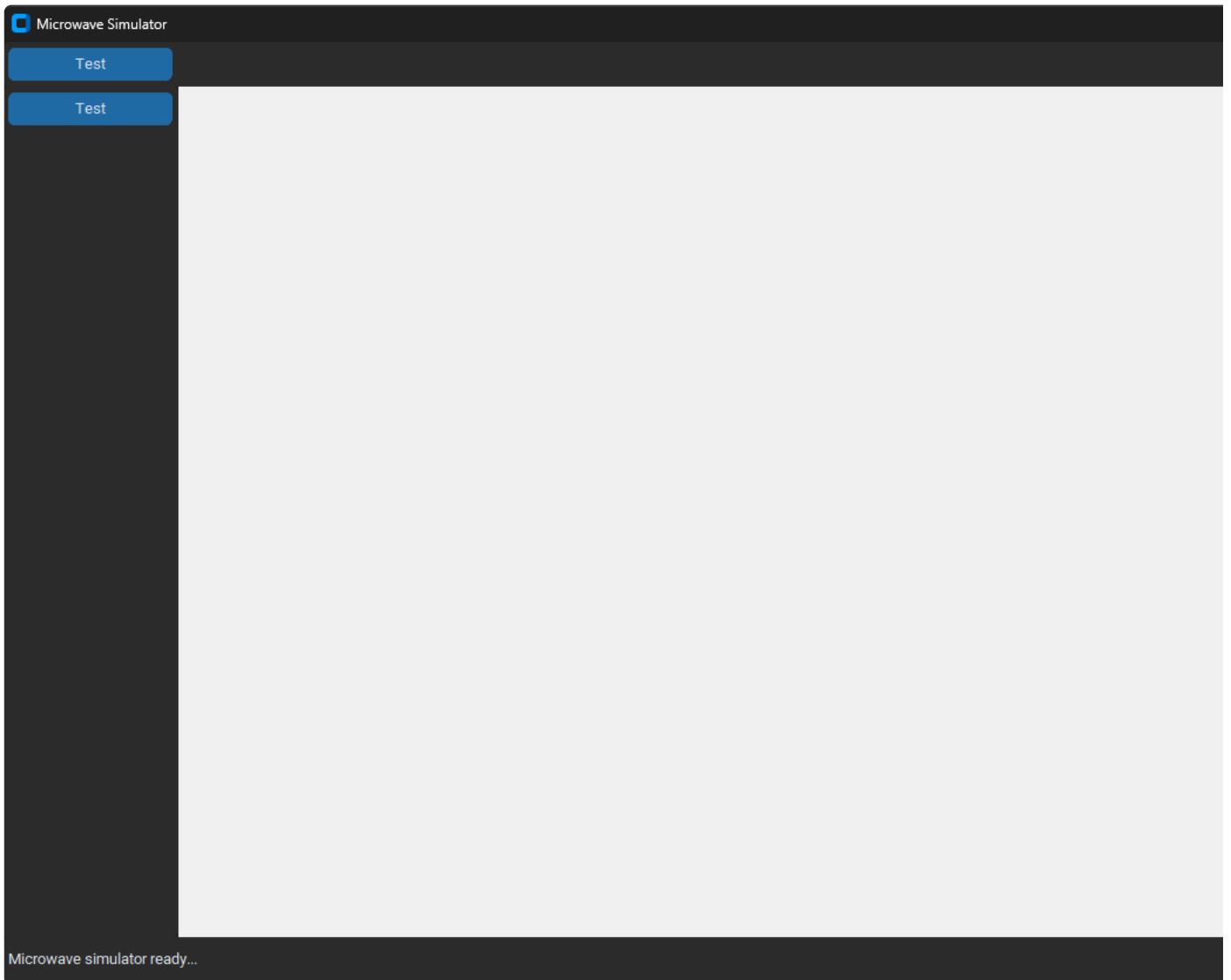


Top Frame

Left
Frame

Canvas

Bottom Frame



microwave_simulator.py

```
import customtkinter as ctk

ctk.set_appearance_mode("System") # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue") # Themes: "blue" (standard), "green",
"dark-blue"

class MicrowaveSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
    window"""

    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
```

```

    self.title("Microwave Simulator")

    self.canvas = ctk.CTkCanvas(self)
    self.top_frame = ctk.CTkFrame(self)
    self.left_frame = ctk.CTkFrame(self)
    self.bottom_frame = ctk.CTkFrame(self)

    self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
    self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
    self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
    self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

    # Add widgets to frames to cause the frames to auto-size
    top_frame_button = ctk.CTkButton(self.top_frame, text="Test")
    top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

    left_frame_button = ctk.CTkButton(self.left_frame, text="Test")
    left_frame_button.pack(side=ctk.TOP, padx=5, pady=5)

    bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Microwave
simulator ready..."))
    bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

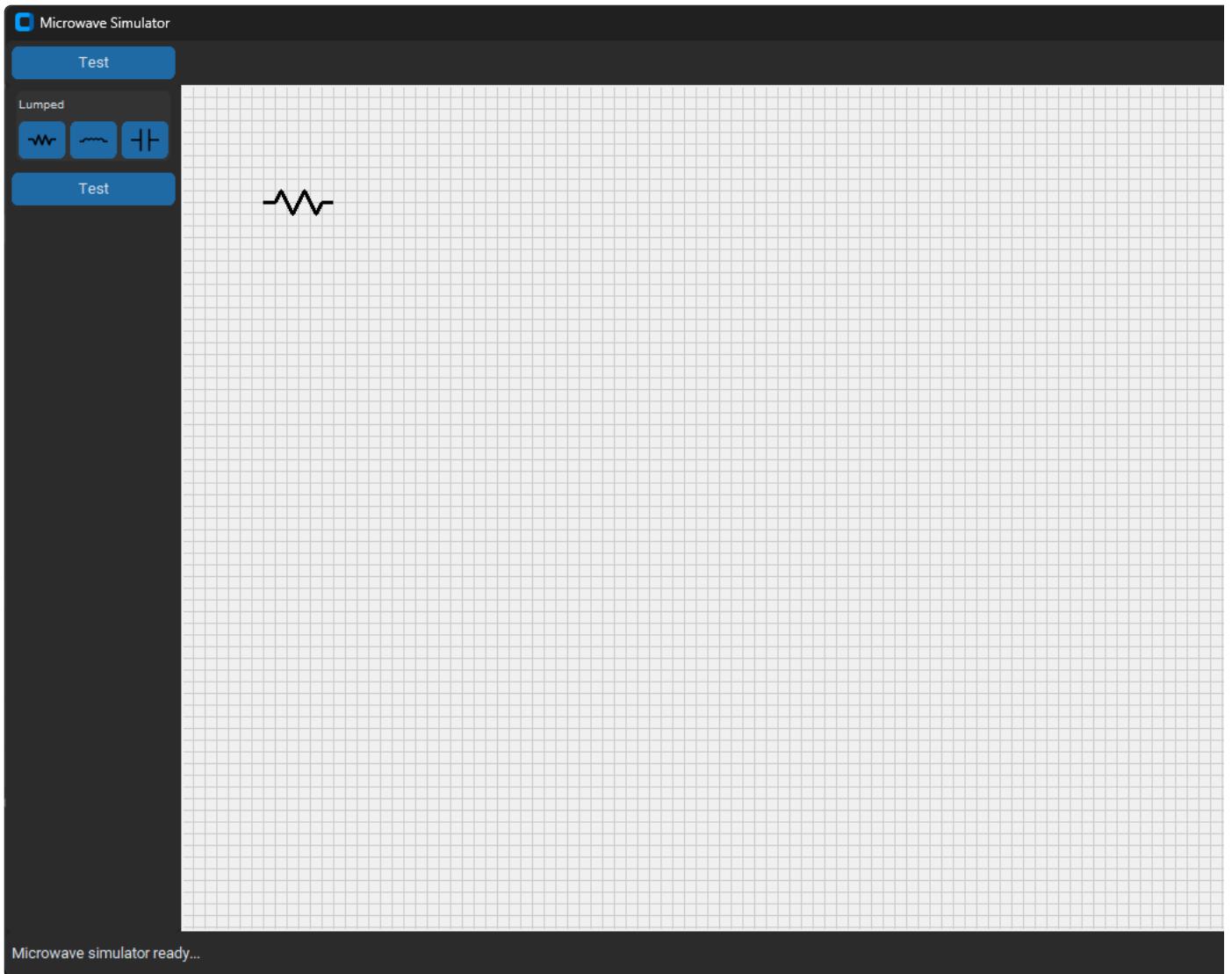
if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = MicrowaveSimulatorApp()
    app.mainloop()

```

Lumped Element Classes

Objectives:

- Create a lumped Resistor Class
- Create a lumped Inductor Class
- Create a lumped Capacitor Class



Comp_Lib/resistor.py

```
class Resistor:  
    """Model for lumped resistor"""\n    def __init__(self, canvas, x1, y1):  
        self.canvas = canvas  
        self.x1, self.y1 = x1, y1  
        self.w = 60  
        self.h = 40  
  
        self.type = "resistor"  
        self.id = None  
        self.bbox = None  
        self.points = []  
  
        self.create_resistor()  
        self.update_bbox()
```

```

def create_resistor(self):
    # Assume comp center is at x1, y1
    self.points = [
        self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 10, self.y1,  #
input lead
        self.x1 - self.w / 2 + 15, self.y1 - 10, self.x1 - self.w/2 + 25,
self.y1 + 10, # Bump 1
        self.x1 - self.w / 2 + 35, self.y1 - 10, self.x1 - self.w / 2 + 45,
self.y1 + 10, # Bump 2
        self.x1 + self.w/2 - 10, self.y1, self.x1 + self.w/2, self.y1 # 
output lead
    ]
    self.id = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')

def update(self):
    pass

def update_bbox(self):
    """Update the bounding box to get current gate coordinates"""
    self.bbox = self.canvas.bbox(self.id)

```

Comp_Lib/inductor.py

```

import tkinter as tk

class Inductor:
    """Model for lumped inductor"""
    def __init__(self, canvas, x1, y1):
        self.canvas = canvas
        self.x1, self.y1 = x1, y1
        self.w = 60
        self.h = 40

        self.type = "inductor"
        self.id = None
        self.bbox = None
        self.ids = []
        self.points = []

        self.create_inductor()


```

```

        self.update_bbox()

def create_inductor(self):
    # Assume comp center is at x1, y1
    self.points = [
        self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 10, self.y1,  #
        input lead
        self.x1 - self.w/2 + 10, self.y1 - 10
    ]
    self.id = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')
    self.ids.append(self.id)

    self.points = [
        self.x1 + self.w / 2 - 10, self.y1 - 10,
        self.x1 + self.w / 2 - 10, self.y1, self.x1 + self.w / 2, self.y1
    # output lead
]
    self.id = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')
    self.ids.append(self.id)

    self.create_lines()
    self.create_arcs()

def create_lines(self):
    for x in range(2, 5):
        self.points = [
            self.x1 - self.w / 2 + 10 * x, self.y1 - 10,
            self.x1 - self.w / 2 + 10 * x, self.y1
        ]
        self.id = self.canvas.create_line(self.points, fill="black",
width=3, tags='comp')
        self.ids.append(self.id)

def create_arcs(self):
    for x in range(2, 6):
        self.points = [
            self.x1 - self.w / 2 + 10 * x, self.y1 - 12,
            self.x1 - self.w / 2 + 10 * (x-1), self.y1 - 2
        ]
        self.id = self.canvas.create_arc(self.points, start=0, extent=180,
fill="black", width=3,
style=tk.ARC, tags='comp')
        self.ids.append(self.id)

```

```

def update(self):
    pass

def update_bbox(self):
    """Update the bounding box to get current gate coordinates"""
    self.bbox = self.canvas.bbox(self.id)

```

Comp_Lib/capacitor.py

```

class Capacitor:
    """Model for lumped capacitor"""
    def __init__(self, canvas, x1, y1):
        self.canvas = canvas
        self.x1, self.y1 = x1, y1
        self.w = 60
        self.h = 40

        self.type = "capacitor"
        self.id = None
        self.bbox = None
        self.points = []
        self.ids = []

        self.create_capacitor()
        self.update_bbox()

    def create_capacitor(self):
        # Assume comp center is at x1, y1
        self.points = [
            self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 25, self.y1,  #
            input lead
            self.x1 - self.w / 2 + 25, self.y1 - 15, self.x1 - self.w/2 + 25,
            self.y1 + 15,
        ]
        self.id = self.canvas.create_line(self.points, fill="black", width=3,
                                         tags='comp')
        self.ids.append(self.id)

        self.points = [
            self.x1 + self.w/2, self.y1, self.x1 + self.w/2 - 25, self.y1,  #
            output lead
            self.x1 + self.w / 2 - 25, self.y1 - 15, self.x1 + self.w/2 - 25,
            self.y1 + 15,

```

```

        ]
        self.id = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')
        self.ids.append(self.id)

    def update(self):
        pass

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

```

UI_Lib/lump_button_frame.py

```

import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from Comp_Lib import Resistor, Inductor, Capacitor


class LumpButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None

        self.button_list = [("resistor", "../icons/resistor.png"),
                           ("inductor", "../icons/inductor.png"),
                           ("capacitor", "../icons/capacitor.png")]

        self.init_frame_widgets()

    def init_frame_widgets(self):
        frame_name_label = ctk.CTkLabel(self, text="Lumped", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        row_num, col_num = 1, 0
        for button in self.button_list:

```

```

    a_image = ctk.CTkImage(light_image=Image.open
                           (Path(__file__).parent / button[1]),
                           dark_image=Image.open
                           (Path(__file__).parent / button[1]),
                           size=(24, 24))
    self.button_id = ctk.CTkButton(self, text="", image=a_image,
width=30,
                           command=lambda
a_name=button[0]:self.create_events(a_name))
    self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W,
padx=2, pady=2)
    ToolTip(self.button_id, msg=button[0])
    row_num, col_num = self.update_grid_numbers(row_num, col_num)

def create_events(self, name):
    gate = None
    if name == "resistor":
        gate = Resistor(self.canvas, 100, 100)
    elif name == "inductor":
        gate = Inductor(self.canvas, 100, 100)
    elif name == "capacitor":
        gate = Capacitor(self.canvas, 100, 100)
    self.canvas.comp_list.append(gate)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

@staticmethod
def update_grid_numbers(row, column):
    column += 1
    if column > 2:
        column = 0
        row += 1
    return row, column

```

UI_Lib/left_frame.py

```

import customtkinter as ctk
from UI_Lib import LumpButtonFrame

class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)

```

```
self.canvas = canvas

self.comp_button_frame = LumpButtonFrame(self, self.canvas)
self.comp_button_frame.pack(side=ctk.TOP, padx=5, pady=5)
```

microwave_simulator.py

```
import customtkinter as ctk

from UI_Lib import LeftFrame, Canvas

ctk.set_appearance_mode("System") # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue") # Themes: "blue" (standard), "green",
"dark-blue"

class MicrowaveSimulatorApp(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""

    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Microwave Simulator")

        self.canvas = Canvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = LeftFrame(self, self.canvas)
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add widgets to frames to cause the frames to auto-size
        top_frame_button = ctk.CTkButton(self.top_frame, text="Test")
        top_frame_button.pack(side=ctk.LEFT, padx=5, pady=5)

        left_frame_button = ctk.CTkButton(self.left_frame, text="Test")
        left_frame_button.pack(side=ctk.TOP, padx=5, pady=5)
```

```

        bottom_frame_label = ctk.CTkLabel(self.bottom_frame, text="Microwave
simulator ready...")
        bottom_frame_label.pack(side=ctk.LEFT, padx=5, pady=5)

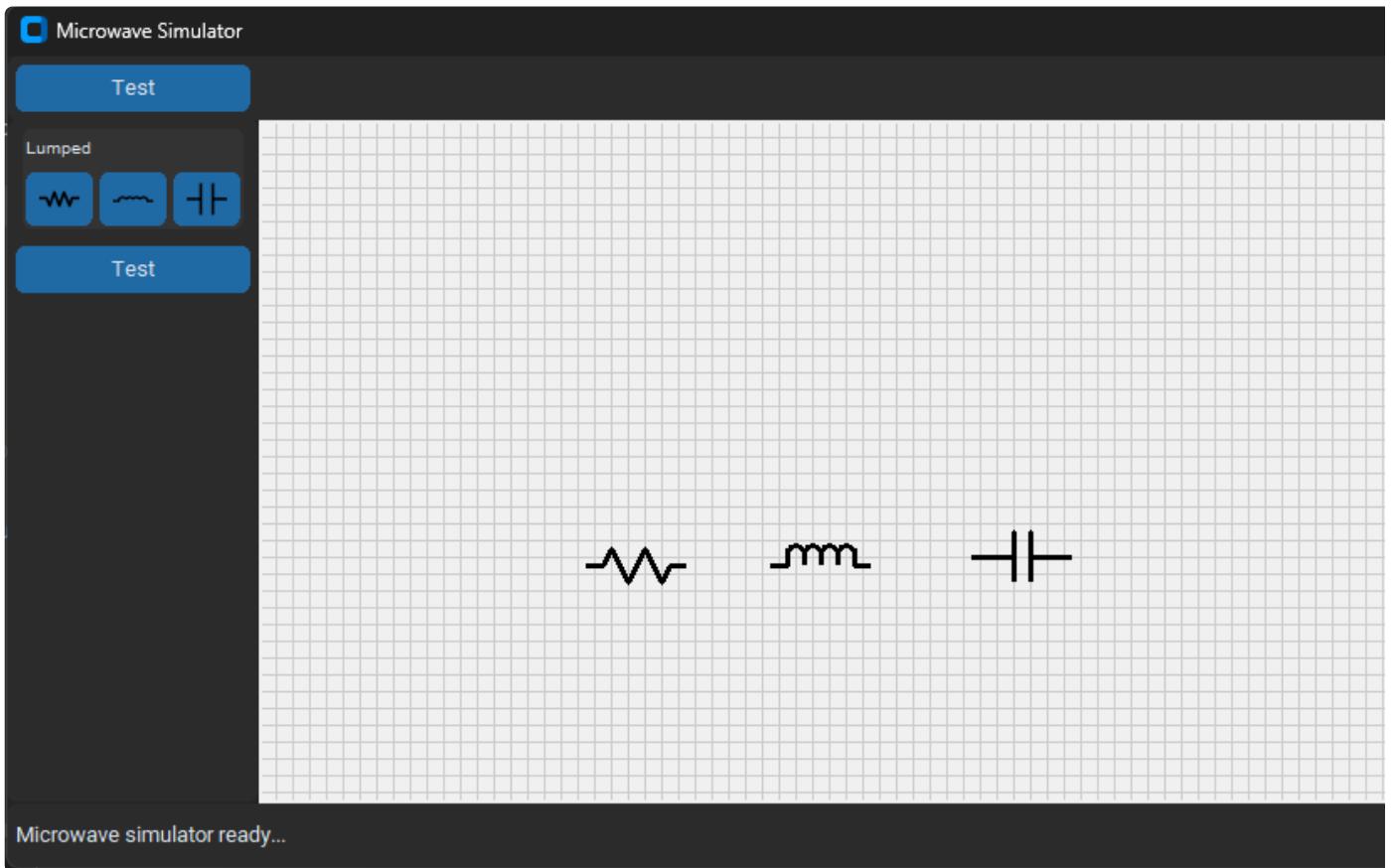
if __name__ == "__main__":
    """Instantiate the Digital Simulator app and run the main loop"""
    app = MicrowaveSimulatorApp()
    app.mainloop()

```

Move Lumped Elements with Mouse

Objectives:

- Add selectors to lumped element classes
- Create a base class for lumped elements
- Add update methods to lumped element classes



Comp_Lib/component.py

```
import tkinter as tk
from PIL import Image, ImageTk


class Comp:
    def __init__(self, canvas, x1, y1):
        """Base class for gate classes"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None
        self.angle = 0
        self.filename = None

        self.is_selected = False
        self.is_drawing = False

        self.in1, self.in2, self.out = None, None, None
        self.conn_list = []
        self.wire_list = []

    def update_position(self):
        """Update the position when the gate object is moved"""
        self.canvas.coords(self.id, self.x1, self.y1) # Update position

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5, self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None, outline="red", width=2)
        self.set_selector_visibility()

    def update_selector(self):
        """Update the red rectangle selector coordinates and check to see if the gate is selected"""



```

```

x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
self.canvas.coords(self.sel_id, x1, y1, x2, y2)
self.set_selector_visibility()

def set_selector_visibility(self):
    """Set the selector visibility state"""
    if self.is_selected:
        self.canvas.itemconfig(self.sel_id, state='normal')
    else:
        self.canvas.itemconfig(self.sel_id, state='hidden')

def set_connector_visibility(self):
    """Set the connector visibility state"""
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

def rotate(self):
    """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
if angle > 270 deg"""
    self.angle += 90
    if self.angle > 270:
        self.angle = 0

def check_connector_hit(self, x, y):
    """Hit test to see if a connector is at the provided x, y
coordinates"""
    for conn in self.conn_list:
        if conn.connector_hit_test(x, y):
            return conn
    return None

def move_connected_wires(self):
    """Resize connected wires if the shape is moved"""
    for connection in self.wire_list:
        for connector in self.conn_list:
            if connector == connection.connector_obj:
                # print(connector, connection.line_obj, "Match")
                if connection.wire_end == "begin":
                    connection.wire_obj.x1 = connector.x
                    connection.wire_obj.y1 = connector.y
                elif connection.wire_end == "end":

```

```
connection.wire_obj.x2 = connector.x
connection.wire_obj.y2 = connector.y
```

Comp_Lib/resistor.py

```
from Comp_Lib.component import Comp


class Resistor(Comp):
    """Model for lumped resistor"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.w = 60
        self.h = 40

        self.type = "resistor"
        self.points = []

        self.create_resistor()
        self.update_bbox()
        self.create_selector()

    def create_resistor(self):
        # Assume comp center is at x1, y1
        self.points = [
            self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 10, self.y1,  #
            input lead
            self.x1 - self.w / 2 + 15, self.y1 - 10, self.x1 - self.w/2 + 25,
            self.y1 + 10,  # Bump 1
            self.x1 - self.w / 2 + 35, self.y1 - 10, self.x1 - self.w / 2 + 45,
            self.y1 + 10,  # Bump 2
            self.x1 + self.w/2 - 10, self.y1, self.x1 + self.w/2, self.y1  #
            output lead
        ]
        self.id = self.canvas.create_line(self.points, fill="black", width=3,
                                         tags='comp')

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_selector()

    def update_position(self):
```

```

        self.points = [
            self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 10, self.y1,  #
            input lead
            self.x1 - self.w / 2 + 15, self.y1 - 10, self.x1 - self.w/2 + 25,
            self.y1 + 10, # Bump 1
            self.x1 - self.w / 2 + 35, self.y1 - 10, self.x1 - self.w / 2 + 45,
            self.y1 + 10, # Bump 2
            self.x1 + self.w/2 - 10, self.y1, self.x1 + self.w/2, self.y1  #
            output lead
        ]
        self.canvas.coords(self.id, self.points)

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = [self.x1 - self.w/2, self.y1 - self.h/2, self.x1 +
        self.w/2, self.y1 + self.h/2]

```

Comp_Lib/inductor.py

```

import tkinter as tk
from Comp_Lib.component import Comp

class Inductor(Comp):
    """Model for lumped inductor"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.w = 60
        self.h = 40

        self.type = "inductor"
        self.points = []
        self.id1, self.id2, self.id3, self.id4, self.id5, self.id6, self.id7,
        self.id8, self.id9 = (
            None, None, None, None, None, None, None, None, None, )

        self.create_inductor()
        self.update_bbox()
        self.create_selector()

    def create_inductor(self):
        # Assume comp center is at x1, y1
        self.points = [

```

```

        self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 10, self.y1, #
input lead
        self.x1 - self.w/2 + 10, self.y1 - 10
    ]
    self.id1 = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')

self.points = [
    self.x1 + self.w / 2 - 10, self.y1 - 10,
    self.x1 + self.w / 2 - 10, self.y1, self.x1 + self.w / 2, self.y1
# output lead
]
self.id2 = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')

self.create_lines()
self.create_arcs()

def create_lines(self):
    self.points = [
        self.x1 - self.w / 2 + 10 * 2, self.y1 - 10,
        self.x1 - self.w / 2 + 10 * 2, self.y1
    ]
    self.id3 = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')

    self.points = [
        self.x1 - self.w / 2 + 10 * 3, self.y1 - 10,
        self.x1 - self.w / 2 + 10 * 3, self.y1
    ]
    self.id4 = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')

    self.points = [
        self.x1 - self.w / 2 + 10 * 4, self.y1 - 10,
        self.x1 - self.w / 2 + 10 * 4, self.y1
    ]
    self.id5 = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')

def create_arcs(self):
    self.points = [
        self.x1 - self.w / 2 + 10 * 2, self.y1 - 12,
        self.x1 - self.w / 2 + 10 * (2-1), self.y1 - 2
    ]
    self.id6 = self.canvas.create_arc(self.points, start=0, extent=180,

```

```

fill="black", width=3, style=tk.ARC,
tags='comp')

self.points = [
    self.x1 - self.w / 2 + 10 * 3, self.y1 - 12,
    self.x1 - self.w / 2 + 10 * (3-1), self.y1 - 2
]
self.id7 = self.canvas.create_arc(self.points, start=0, extent=180,
                                    fill="black", width=3, style=tk.ARC,
tags='comp')

self.points = [
    self.x1 - self.w / 2 + 10 * 4, self.y1 - 12,
    self.x1 - self.w / 2 + 10 * (4-1), self.y1 - 2
]
self.id8 = self.canvas.create_arc(self.points, start=0, extent=180,
                                    fill="black", width=3, style=tk.ARC,
tags='comp')

self.points = [
    self.x1 - self.w / 2 + 10 * 5, self.y1 - 12,
    self.x1 - self.w / 2 + 10 * (5-1), self.y1 - 2
]
self.id9 = self.canvas.create_arc(self.points, start=0, extent=180,
                                    fill="black", width=3, style=tk.ARC,
tags='comp')

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_selector()

def update_position(self):
    # Update lead positions
    self.points = [
        self.x1 - self.w / 2, self.y1, self.x1 - self.w / 2 + 10, self.y1,
    # input lead
        self.x1 - self.w / 2 + 10, self.y1 - 10
    ]
    self.canvas.coords(self.id1, self.points)

    self.points = [
        self.x1 + self.w / 2 - 10, self.y1 - 10,
        self.x1 + self.w / 2 - 10, self.y1, self.x1 + self.w / 2, self.y1
    # output lead
    ]

```

```

    self.canvas.coords(self.id2, self.points)

    # Update line positions
    self.points = [
        self.x1 - self.w / 2 + 10 * 2, self.y1 - 10,
        self.x1 - self.w / 2 + 10 * 2, self.y1
    ]
    self.canvas.coords(self.id3, self.points)

    self.points = [
        self.x1 - self.w / 2 + 10 * 3, self.y1 - 10,
        self.x1 - self.w / 2 + 10 * 3, self.y1
    ]
    self.canvas.coords(self.id4, self.points)

    self.points = [
        self.x1 - self.w / 2 + 10 * 4, self.y1 - 10,
        self.x1 - self.w / 2 + 10 * 4, self.y1
    ]
    self.canvas.coords(self.id5, self.points)

    # Update arc positions
    self.points = [
        self.x1 - self.w / 2 + 10 * 2, self.y1 - 12,
        self.x1 - self.w / 2 + 10 * (2-1), self.y1 - 2
    ]
    self.canvas.coords(self.id6, self.points)

    self.points = [
        self.x1 - self.w / 2 + 10 * 3, self.y1 - 12,
        self.x1 - self.w / 2 + 10 * (3-1), self.y1 - 2
    ]
    self.canvas.coords(self.id7, self.points)

    self.points = [
        self.x1 - self.w / 2 + 10 * 4, self.y1 - 12,
        self.x1 - self.w / 2 + 10 * (4-1), self.y1 - 2
    ]
    self.canvas.coords(self.id8, self.points)

    self.points = [
        self.x1 - self.w / 2 + 10 * 5, self.y1 - 12,
        self.x1 - self.w / 2 + 10 * (5-1), self.y1 - 2
    ]
    self.canvas.coords(self.id9, self.points)

```

```

def update_bbox(self):
    """Update the bounding box to get current gate coordinates"""
    self.bbox = [self.x1 - self.w/2, self.y1 - self.h/2, self.x1 +
    self.w/2, self.y1 + self.h/2]

```

Comp_Lib/capacitor.py

```

from Comp_Lib.component import Comp

class Capacitor(Comp):
    """Model for lumped capacitor"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.w = 60
        self.h = 40

        self.type = "capacitor"
        self.points = []

        self.id1, self.id2 = None, None

        self.create_capacitor()
        self.update_bbox()
        self.create_selector()

    def create_capacitor(self):
        # Assume comp center is at x1, y1
        self.points = [
            self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 25, self.y1,  #
            input lead
            self.x1 - self.w / 2 + 25, self.y1 - 15, self.x1 - self.w/2 + 25,
            self.y1 + 15,
        ]
        self.id1 = self.canvas.create_line(self.points, fill="black", width=3,
                                         tags='comp')

        self.points = [
            self.x1 + self.w/2, self.y1, self.x1 + self.w/2 - 25, self.y1,  #
            output lead
            self.x1 + self.w / 2 - 25, self.y1 - 15, self.x1 + self.w/2 - 25,
            self.y1 + 15,
        ]

```

```

        self.id2 = self.canvas.create_line(self.points, fill="black", width=3,
tags='comp')

    def update(self):
        self.update_position()
        self.update_bbox()
        self.update_selector()

    def update_position(self):
        self.points = [
            self.x1 - self.w/2, self.y1, self.x1 - self.w/2 + 25, self.y1,  #
input lead
            self.x1 - self.w / 2 + 25, self.y1 - 15, self.x1 - self.w/2 + 25,
self.y1 + 15,
        ]
        self.canvas.coords(self.id1, self.points)

        self.points = [
            self.x1 + self.w/2, self.y1, self.x1 + self.w/2 - 25, self.y1,  #
output lead
            self.x1 + self.w / 2 - 25, self.y1 - 15, self.x1 + self.w/2 - 25,
self.y1 + 15,
        ]
        self.canvas.coords(self.id2, self.points)

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = [self.x1 - self.w/2, self.y1 - self.h/2, self.x1 +
self.w/2, self.y1 + self.h/2]

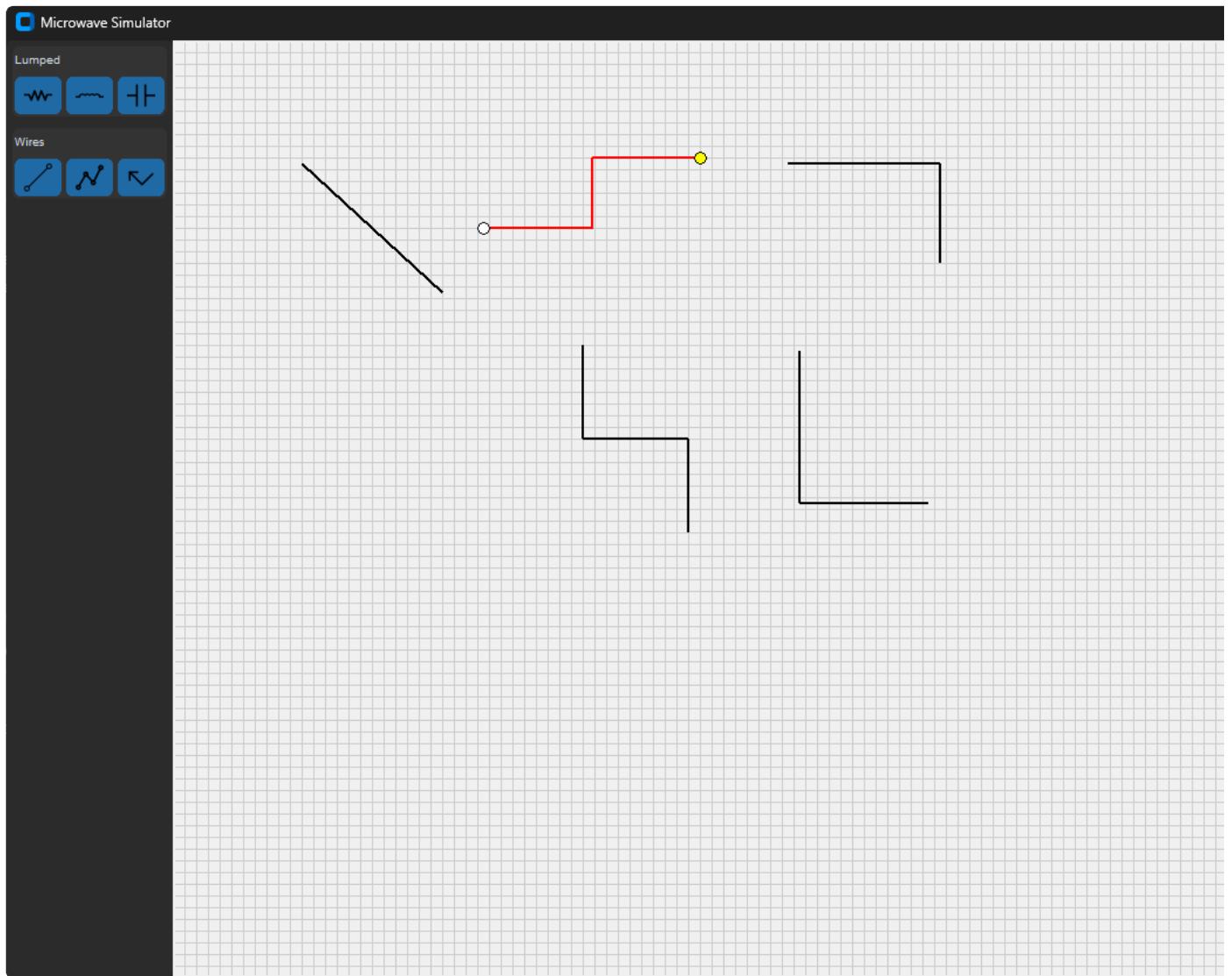
```

Wire Classes

Objectives:

- Straight wire class
 - Straight wire hit test based on equation of a line between two points
- 3-Segment Wire Class
 - Horizontal and vertical orientation based on line direction when drawing
- 2-Segment Elbow Class
 - Horizontal and vertical orientation based on line direction when drawing
- Wire select and move with mouse

- End selector color change when selected
- Wire resize with mouse



Wire_Lib / __init__.py

```
# import wire related classes
from Wire_Lib.straight_wire import StraightWire
from Wire_Lib.segment_wire import SegmentWire
from Wire_Lib.elbow_wire import ElbowWire

# import wire selector related classes
from Wire_Lib.wire_selector import WireSelector
```

Wire_Lib /wire.py - base class for wire classes

```
from Wire_Lib.wire_selector import WireSelector


class Wire:
    """Base class for wire classes"""
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        self.fill_color = "black"
        self.border_width = 2
        self.line_direction = "horizontal"

        self.id = None
        self.is_selected = False
        self.selector = None
        self.width = 2
        self.bbox = None

        self.sel1, self.sel2 = None, None

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def create_selectors(self):
        self.sel1 = WireSelector(self.canvas, "begin", self.x1, self.y1)
        self.sel2 = WireSelector(self.canvas, "end", self.x2, self.y2)

    def update_selectors(self):
        self.sel1.x, self.sel1.y = self.x1, self.y1
        self.sel2.x, self.sel2.y = self.x2, self.y2
        self.sel1.update()
        self.sel2.update()

    def update_selection(self):
        if self.is_selected:
            self.canvas.itemconfigure(self.id, fill="red")
            self.canvas.itemconfigure(self.sel1.id, state='normal')
            self.canvas.itemconfigure(self.sel2.id, state='normal')
        else:
            self.canvas.itemconfigure(self.id, fill="black")
            self.canvas.itemconfigure(self.sel1.id, state='hidden')
            self.canvas.itemconfigure(self.sel2.id, state='hidden')
```

```

def hit_test(self, x, y):
    x1, y1 = self.bbox[0], self.bbox[1]
    x2, y2 = self.bbox[2], self.bbox[3]
    if x1 <= x <= x2 and y1 <= y <= y2:
        self.is_selected = True
    else:
        self.is_selected = False

def sel_hit_test(self, x, y):
    if self.sel1.selector_hit_test(x, y):
        self.selector = self.sel1.name
        return self.sel1
    elif self.sel2.selector_hit_test(x, y):
        self.selector = self.sel2.name
        return self.sel2
    else:
        return None

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
        self.x2, self.y2 = self.canvas.grid.snap_to_grid(self.x2, self.y2)
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
        self.x1, self.y1 = self.canvas.grid.snap_to_grid(self.x1, self.y1)

```

Wire_Lib/straight_wire.py

```

from Wire_Lib.wire import Wire

class StraightWire(Wire):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)

        self.create_wire()
        self.create_selectors()

```

```

        self.update_selection()

    def create_wire(self):
        self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
width=self.width)

    def update(self):
        self.update_position()
        self.update_selectors()
        self.update_selection()

    def update_position(self):
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)

    def hit_test(self, x, y):
        # 2-Point Line equation: y = m * (x - x1) + y1
        x1, y1 = self.x1, self.y1
        x2, y2 = self.x2, self.y2

        # Calculate the slope: m = (y2 - y1) / (x2 - x1)
        if (x2 - x1) == 0:
            m = 0
        else:
            m = (y2 - y1)/(x2 - x1)

        # Check to see if the point (x, y) is on the line and between the two
end points
        tol = 10
        if y - tol <= m*(x - x1) + y1 <= y + tol:
            if (min(x1, x2) <= x <= max(x1, x2)) and (min(y1, y2) <= y <=
max(y1, y2)):
                self.is_selected = True
            else:
                self.is_selected = False

```

Wire_Lib/segment_wire.py

```

from Wire_Lib.wire import Wire

class SegmentWire(Wire):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)

```

```

        self(seg1, self(seg2, self(seg3 = None, None, None
        self.segment_list = []

        self.create_wire()
        self.update_bbox()
        self.create_selectors()
        self.update_selection()

def create_wire(self):
    w = self.x2 - self.x1
    h = self.y2 - self.y1

    if abs(w) >= abs(h): # Horizontal
        self.seg1 = self.x1, self.y1, self.x1 + w / 2, self.y1
        self.seg2 = self.x1 + w / 2, self.y1, self.x1 + w / 2, self.y2
        self.seg3 = self.x1 + w / 2, self.y2, self.x2, self.y2
    else: # Vertical
        self.seg1 = self.x1, self.y1, self.x1, self.y1 + h / 2
        self.seg2 = self.x1, self.y1 + h / 2, self.x2, self.y1 + h / 2
        self.seg3 = self.x2, self.y1 + h / 2, self.x2, self.y2
    self.segment_list = [self.seg1, self.seg2, self.seg3]
    self.draw_segments()

def draw_segments(self):
    for s in self.segment_list:
        self.id = self.canvas.create_line(s, fill=self.fill_color,
width=self.border_width, tags='wire')

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_selectors()
    self.update_selection()

def update_position(self):
    """Update the position when the gate object is moved"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    if abs(w) >= abs(h):
        self.canvas.coords(self.id, self.x1, self.y1, self.x1 + w / 2,
self.y1,
                           self.x1 + w / 2, self.y1, self.x1 + w / 2,
self.y2,
                           self.x1 + w / 2, self.y2, self.x2, self.y2)
    else:

```

```

        self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y1 + h
/ 2,
                         self.x1, self.y1 + h / 2, self.x2, self.y1 + h /
2,
                         self.x2, self.y1 + h / 2, self.x2, self.y2)

```

Wire_Lib/elbow_wire.py

```

from Wire_Lib.wire import Wire


class ElbowWire(Wire):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)

        self.create_wire()
        self.update_bbox()
        self.create_selectors()
        self.update_selection()

    def create_wire(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        if abs(w) >= abs(h): # Horizontal
            self.id = self.canvas.create_line(self.x1, self.y1, self.x2,
self.y1,
                                              self.x2, self.y1, self.x2,
self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width,
tags="wire")
        else: # Vertical
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1,
self.y2,
                                              self.x1, self.y2, self.x2,
self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width,
tags="wire")

    def update(self):
        self.update_position()

```

```

    self.update_bbox()
    self.update_selectors()
    self.update_selection()

def update_position(self):
    """Update the position when the gate object is moved"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    if abs(w) >= abs(h):
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y1,
                           self.x2, self.y1, self.x2, self.y2)
    else:
        self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y2,
                           self.x1, self.y2, self.x2, self.y2)

```

Wire_Lib/wire_selector.py

```

class WireSelector:
    def __init__(self, canvas, name, x, y):
        self.canvas = canvas
        self.name = name
        self.x = x
        self.y = y
        self.is_selected = False

        self.id = self.canvas.create_oval(self.x - 5, self.y - 5, self.x + 5,
                                         self.y + 5,
                                         state='normal', fill="white")

    def update(self):
        self.update_position()
        self.update_selection()

    def update_position(self):
        """Update the selector position"""
        sel_points = [self.x - 5, self.y - 5, self.x + 5, self.y + 5]
        self.canvas.coords(self.id, sel_points)

    def update_selection(self):
        if self.is_selected:
            self.canvas.itemconfigure(self.id, fill="yellow")
        else:
            self.canvas.itemconfigure(self.id, fill="white")

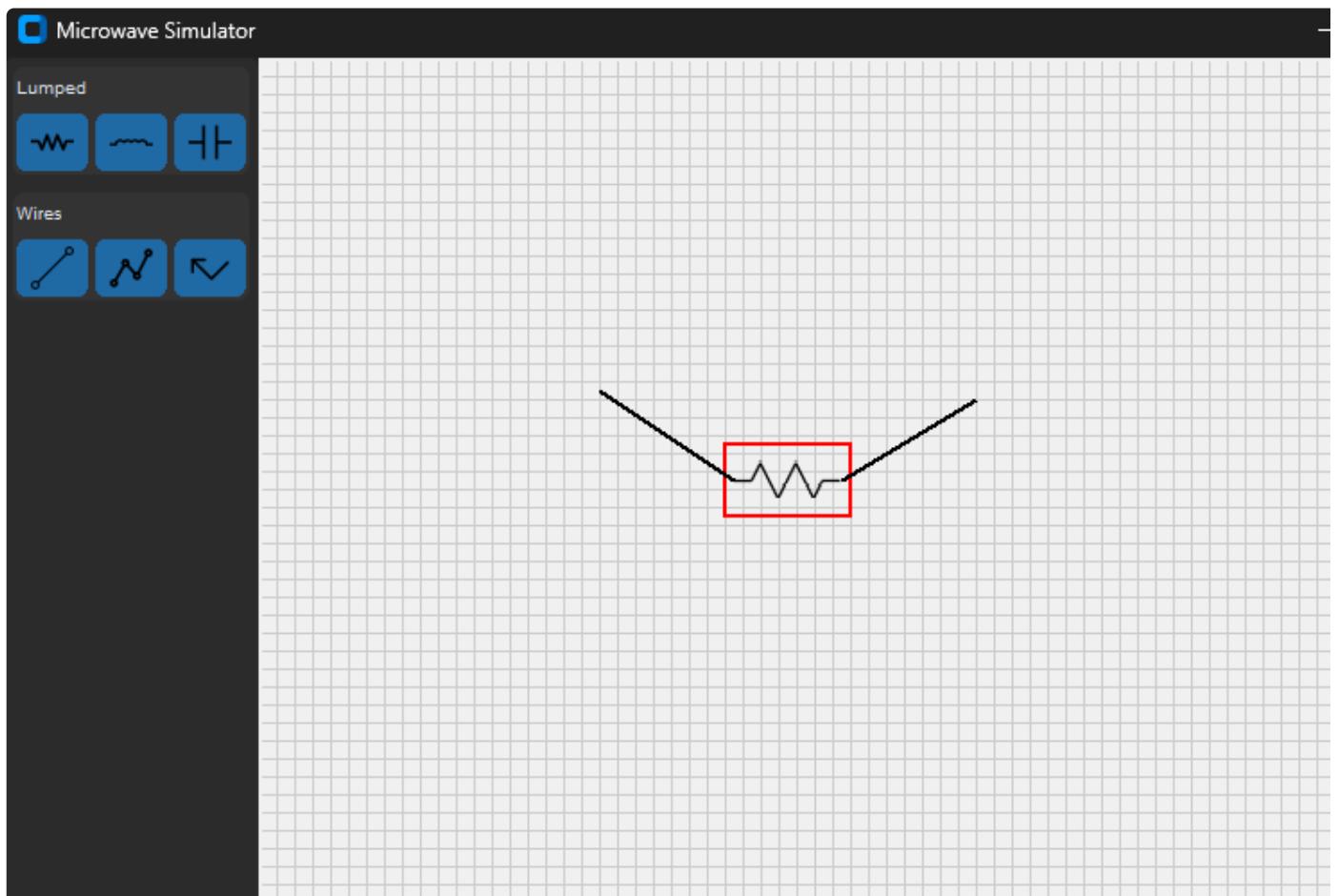
```

```
def selector_hit_test(self, event_x, event_y):
    if self.x-5 <= event_x <= self.x+5 and self.y-5 <= event_y <= self.y+5:
        self.is_selected = True
        self.update_selection()
        return True
    else:
        self.is_selected = False
        self.update_selection()
        return False
```

Connection Classes

Objectives:

- Connect wires to component "connectors"
- Add wire list to each component to keep track of connected wires
- Move connected wires when component moves with mouse



Comp_Lib/ __init__.py

```
# Import lumped components
from Comp_Lib.resistor import Resistor
from Comp_Lib.inductor import Inductor
from Comp_Lib.capacitor import Capacitor

# Import connector related classes
from Comp_Lib.connector import Connector
from Comp_Lib.connection import Connection
```

Comp_Lib/connector.py

```
class Connector:  
    def __init__(self, canvas, name, x, y):  
        """Connector class"""  
        self.canvas = canvas  
        self.name = name  
        self.x = x  
        self.y = y  
  
        self.id = None  
  
        self.radius = 5  
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -  
self.radius,  
                                         self.x + self.radius, self.y +  
self.radius)  
  
        self.create_connector()  
  
    def create_connector(self):  
        # Create the connector here  
        points = [self.x - self.radius, self.y - self.radius, self.x +  
self.radius, self.y + self.radius]  
        self.id = self.canvas.create_oval(points, fill="white",  
outline="black", width=2, tags='connector')  
  
    def update(self):  
        """Update the connector here"""  
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -  
self.radius,
```

```

                self.x + self.radius, self.y +
self.radius)
    points = [self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius]
    self.canvas.coords(self.id, points)

def set_pos(self, x, y):
    """Set the connector position here"""
    self.x = x
    self.y = y

def connector_hit_test(self, x, y):
    """Connector hit test"""
    if self.x1 <= x <= self.x2 and self.y1 <= y <= self.y2:
        return True
    else:
        return False

def __repr__(self):
    return ("Connector: " + self.name + " (" + str(self.x1) + ", " +
str(self.y1) + ")" +
        " (" + str(self.x2) + ", " + str(self.y2) + ")")

```

Comp_Lib/connection.py

```

class Connection:
    def __init__(self, conn_obj, wire_obj, wire_end):
        self.connector_obj = conn_obj
        self.wire_obj = wire_obj
        self.wire_end = wire_end      # "begin" or "end"

    def __repr__(self):
        return "Connection Object: " + str(self.connector_obj.name) + \
               " Connection Object Location: " + str(self.connector_obj.x) + ", " +
str(self.connector_obj.y) + \
               " Line Object: " + str(self.wire_obj) + \
               " Line End: " + self.wire_end

```

Comp_Lib/component.py

```

import tkinter as tk
from PIL import Image, ImageTk

from Helper_Lib import Point # Import Point Class
from Comp_Lib.connector import Connector # Import Connector Class


class Component:
    def __init__(self, canvas, x1, y1):
        """Base class for component classes"""
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.id = None
        self.sel_id = None
        self.is_selected = False
        self.is_drawing = False
        self.selector = None
        self.angle = 0

        self.a_image = None
        self.ph_image = None
        self.bbox = None

        self.conn_list = [] # Connector list
        self.wire_list = [] # Wire list (list of connections)
        . . .

    def create_connectors(self): # Connector method
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        # Define 3 connectors: in1, in2, out
        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y)

        # Update the connector list
        self.conn_list = [self.out, self.in1]

    def update_connectors(self): # Connector method
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1

```



```
        connection.wire_obj.y1 = connector.y
    elif connection.wire_end == "end":
        connection.wire_obj.x2 = connector.x
        connection.wire_obj.y2 = connector.y

    . . .
```

Comp_Lib/resistor.py

```
from pathlib import Path

from Helper_Lib import Point
from Comp_Lib.component import Component
from Comp_Lib.connector import Connector # import connector class

class Resistor(Component):
    """Model for lumped element resistor"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent /
"../images/lumped/resistor_60x30.png"

        self.in1, self.out = None, None
        self.conn_list = []

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors() # Connector method
        self.set_connector_visibility() # Connector method

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors() # Connector method
        self.set_connector_visibility() # Connector method
```

Comp_Lib/inductor.py

```

from pathlib import Path

from Comp_Lib.component import Component


class Inductor(Component):
    """Model for lumped element inductor"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent /
"../images/lumped/inductor_60x30.png"
        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()

    # Create connectors
    self.in1, self.out = None, None
    self.conn_list = []
    self.create_connectors()
    self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

```

Comp_Lib/capacitor.py

```

from pathlib import Path

from Comp_Lib.component import Component


class Capacitor(Component):
    """Model for lumped element capacitor"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent /
"../images/lumped/capacitor_60x30.png"
        self.create_image(self.filename)

```

```

    self.update_bbox()
    self.create_selector()

    # Create connectors
    self.in1, self.out = None, None
    self.conn_list = []
    self.create_connectors()
    self.set_connector_visibility()

def update(self):
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

```

UI.Lib/canvas.py

```

import customtkinter as ctk

from UI.Lib.mouse import Mouse
from UI.Lib.grid import Grid


class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

        self.led_color = "red"
        self.led_size = "large"
        self.grid = Grid(self, 10)
        self.mouse = Mouse(self)
        self.comp_list = []

        self.mouse.move_mouse_bind_events()

    def redraw(self):
        self.delete('grid_line')
        self.grid.draw()
        self.tag_lower("grid_line")
        for c in self.comp_list:
            c.update()

```

```

def show_connectors(self): # Show all component connectors
    for s in self.comp_list:
        s.is_drawing = True
    self.redraw()

def hide_connectors(self): # Hide all component connector
    for s in self.comp_list:
        s.is_drawing = False
    self.redraw()

def edit_shape(self, _event=None):
    pass

```

UI_Lib/mouse.py

```

from Helper_Lib import Point
from Wire_Lib import StraightWire, SegmentWire, ElbowWire
from Comp_Lib import Connection # Import connection class

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas

        self.selected_comp = None
        self.current_wire_obj = None
        self.start = Point(0, 0)
        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

        self.move_mouse_bind_events()

    . . .

    def draw_left_down(self, event): # Added method for draw left down
        if self.current_wire_obj:
            # self.unselect_all()
            self.start.x = event.x
            self.start.y = event.y
            self.start.x, self.start.y =
                self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

```

```

        self.current_wire_obj.x1, self.current_wire_obj.y1 = self.start.x,
self.start.y
        self.current_wire_obj.x2, self.current_wire_obj.y2 = self.start.x,
self.start.y

        self.select_connector(self.current_wire_obj, "begin", self.start.x,
self.start.y) # Check for connector hit

def draw_left_drag(self, event): # Added method for draw left drag
    if self.current_wire_obj:
        wire = self.current_wire_obj
        x, y = event.x, event.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        wire.x1, wire.y1 = self.start.x, self.start.y
        wire.x2, wire.y2 = x, y
        self.current_wire_obj.update()

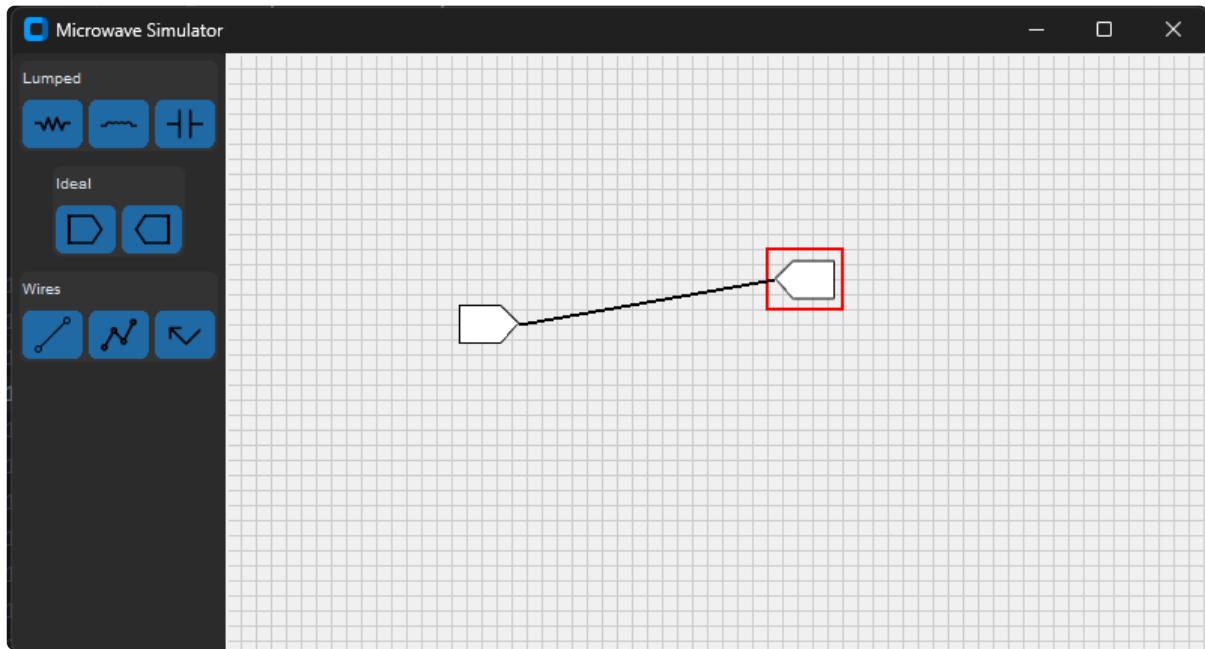
def draw_left_up(self, event): # Added method for draw left up
    self.select_connector(self.current_wire_obj, "end", event.x, event.y)
# Check for connector hit
    self.canvas.hide_connectors() # Hide connectors
    self.current_wire_obj = None
    self.move_mouse_bind_events()

    . . .

def select_connector(self, wire_obj, wire_end, x, y): # Added method to
see if line end hits a comp connector
    for comp in self.canvas.comp_list:
        if not (isinstance(comp, StraightWire) or isinstance(comp,
SegmentWire) or
                isinstance(self.selected_comp, ElbowWire)):
            conn = comp.check_connector_hit(x, y)
            if conn:
                if wire_end == "begin":
                    wire_obj.x1, wire_obj.y1 = conn.x, conn.y
                elif wire_end == "end":
                    wire_obj.x2, wire_obj.y2 = conn.x, conn.y
                a_conn = Connection(conn, self.current_wire_obj, wire_end)
                comp.wire_list.append(a_conn)
                self.canvas.redraw()

```

Port Classes



Comp_Lib/import.py

```
from pathlib import Path

from Comp_Lib.component import Component
from Helper_Lib import Point
from Comp_Lib.connector import Connector


class Inport(Component):
    """Model for input port"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/ports/in_port.png"

        self.in1, self.out = None, None
        self.conn_list = []

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
```

```

        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self): # Override base class method
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.out = Connector(self.canvas, "out", center.x + w / 2, center.y)

        # Update the connector list
        self.conn_list = [self.out]

    def update_connectors(self): # Override base class method
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.out.x, self.out.y = center.x + w / 2, center.y
        elif self.angle == 90:
            self.out.x, self.out.y = center.x, center.y - h / 2
        elif self.angle == 180:
            self.out.x, self.out.y = center.x - w / 2, center.y
        elif self.angle == 270:
            self.out.x, self.out.y = center.x, center.y + h / 2

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

```

Comp_Lib/outport.py

```

from pathlib import Path

from Comp_Lib.component import Component
from Helper_Lib import Point
from Comp_Lib.connector import Connector

```

```

class Outport(Component):
    """Model for output port"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/ports/out_port.png"

        self.in1, self.out = None, None
        self.conn_list = []

        self.create_image(self.filename)
        self.update_bbox()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self): # Override base class method
        # Calculate position of connectors from current shape position and size
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w, h = x2 - x1, y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        self.in1 = Connector(self.canvas, "in1", center.x - w / 2, center.y)

        # Update the connector list
        self.conn_list = [self.in1]

    def update_connectors(self): # Override base class method
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.angle == 0:
            self.in1.x, self.in1.y = center.x - w / 2, center.y
        elif self.angle == 90:

```

```
    self.in1.x, self.in1.y = center.x, center.y + h / 2
elif self.angle == 180:
    self.in1.x, self.in1.y = center.x + w / 2, center.y
elif self.angle == 270:
    self.in1.x, self.in1.y = center.x, center.y - h / 2

for c in self.conn_list:
    c.update()

self.move_connected_wires()
```

UI_Lib/ideal_button_frame.py

```

        dark_image=Image.open
        (Path(__file__).parent / button[1]),
        size=(24, 24))
    self.button_id = ctk.CTkButton(self, text="", image=a_image,
width=30,
                           command=lambda
a_name=button[0]:self.create_events(a_name))
    self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W,
padx=2, pady=2)
    ToolTip(self.button_id, msg=button[0])
    row_num, col_num = self.update_grid_numbers(row_num, col_num)

def create_events(self, name):
    comp = None
    if name == "inport":
        comp = Inport(self.canvas, 100, 100)
    elif name == "outport":
        comp = Outport(self.canvas, 100, 100)
    self.canvas.comp_list.append(comp)
    self.canvas.redraw()
    self.canvas.mouse_move_bind_events()

@staticmethod
def update_grid_numbers(row, column):
    column += 1
    if column > 2:
        column = 0
        row += 1
    return row, column

```

UI_Lib/left_frame.py

```

import customtkinter as ctk
from UI_Lib import LumpButtonFrame, IdealButtonFrame, WireButtonFrame

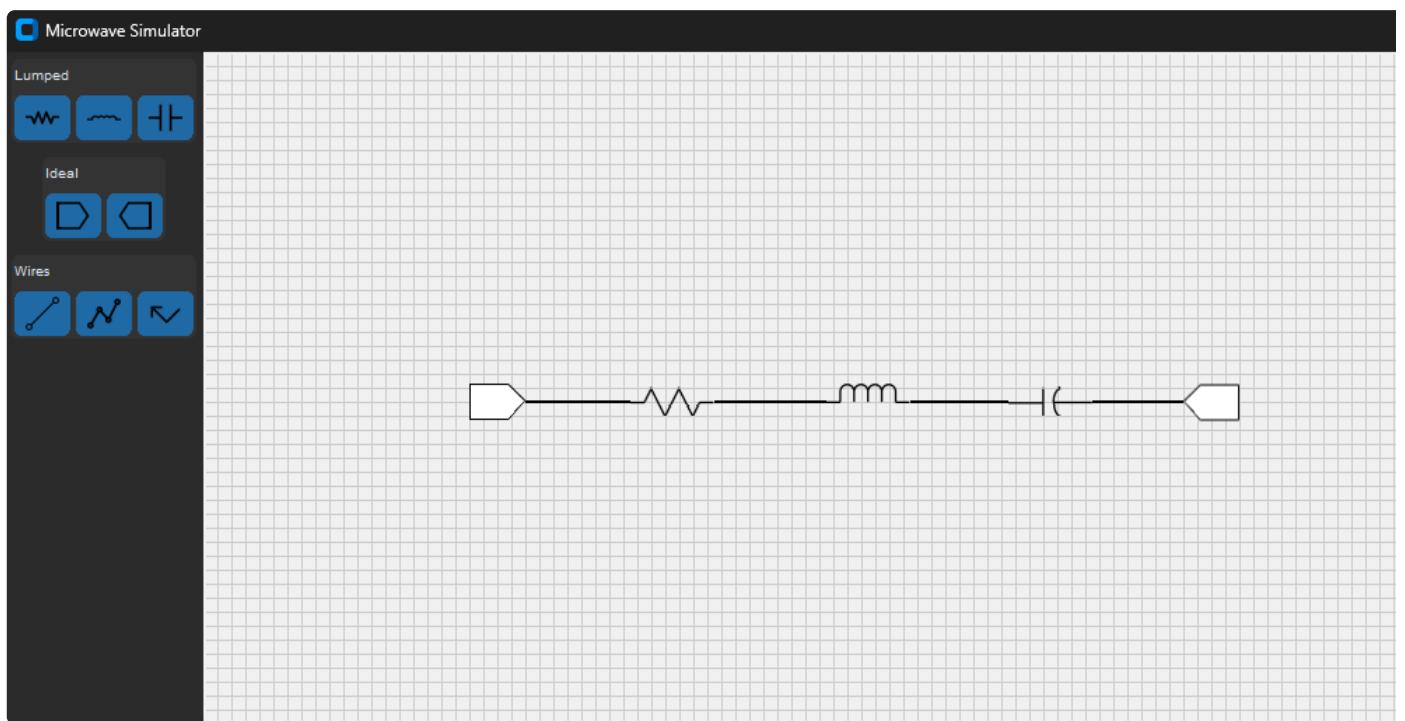
class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

        self.comp_button_frame = LumpButtonFrame(self, self.canvas)
        self.comp_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

```

```
    self.ideal_button_frame = IdealButtonFrame(self, self.canvas) # Added  
ideal button frame  
    self.ideal_button_frame.pack(side=ctk.TOP, padx=5, pady=5)  
  
    self.wire_button_frame = WireButtonFrame(self, self.canvas)  
    self.wire_button_frame.pack(side=ctk.TOP, padx=5, pady=5)
```

RLC Circuit

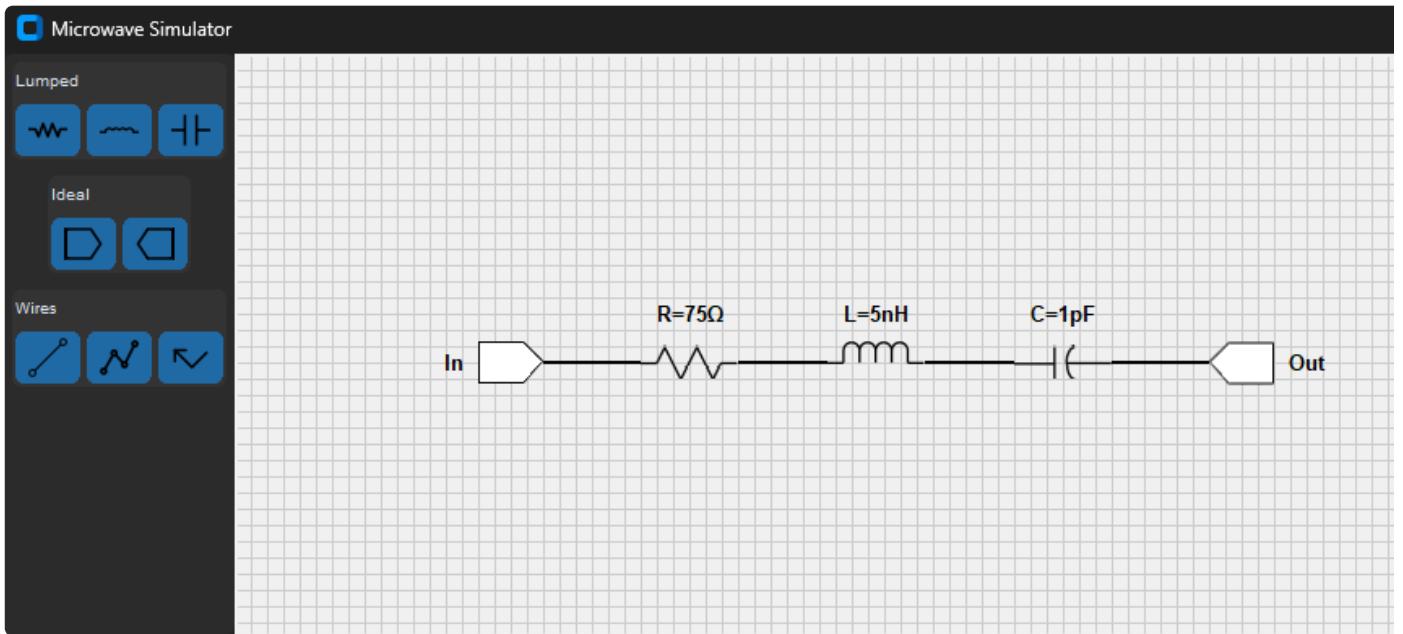


Circuit drawn without any additional code changes.

Objectives:

- Add text to components - show component values
- Save circuit
- Load circuit

Add Text to Components



Comp_Lib/inport.py

```

from pathlib import Path

from Comp_Lib.component import Component
from Helper_Lib import Point
from Comp_Lib.connector import Connector


class Inport(Component):
    """Model for input port"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/ports/in_port.png"

        self.in1, self.out = None, None
        self.conn_list = []

        self.text = "In" # Add text attribute
        self.text_id = None # Id for text item

        self.create_image(self.filename)
        self.update_bbox()
        self.create_text() # New method call to create text
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

```

```

def update(self):
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_text() # New method call to update text
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def create_text(self): # New method to create text
    self.text_id = self.canvas.create_text(self.x1-35, self.y1,
                                           text=self.text, fill="black",
                                           font='Helvetica 10 bold',
                                           angle=self.angle, tags="text")

def update_text(self): # New Method to update text
    self.canvas.coords(self.text_id, self.x1-35, self.y1)

    . . .

```

Comp_Lib/output.py

```

from pathlib import Path

from Comp_Lib.component import Component
from Helper_Lib import Point
from Comp_Lib.connector import Connector


class Outport(Component):
    """Model for output port"""
    def __init__(self, canvas, x1, y1):
        super().__init__(canvas, x1, y1)
        self.filename = Path(__file__).parent / "../images/ports/out_port.png"

        self.in1, self.out = None, None
        self.conn_list = []

        self.text = "Out" # Add text attribute
        self.text_id = None # Id for text item

        self.create_image(self.filename)
        self.update_bbox()

```

```

        self.create_text() # New method call to create text
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_text() # New method call to update text
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_text(self): # New method to create text
        self.text_id = self.canvas.create_text(self.x1+40, self.y1,
                                              text=self.text, fill="black",
                                              font='Helvetica 10 bold',
                                              angle=self.angle, tags="text")

    def update_text(self): # New Method to update text
        self.canvas.coords(self.text_id, self.x1+40, self.y1)

    . . .

```

Comp_Lib/resistor.py

```

from pathlib import Path

from Comp_Lib.component import Component


class Resistor(Component):
    """Model for lumped element resistor"""
    def __init__(self, canvas, x1, y1, resistance): # Added resistance
        argument
        super().__init__(canvas, x1, y1)
        self.resistance = resistance # Added resistance attribute
        self.filename = Path(__file__).parent /
        "../images/lumped/resistor_60x30.png"

        self.in1, self.out = None, None
        self.conn_list = []

```

```

        self.text = 'R=' + str(resistance) + '\u2126' # Create resistor text
using resistance
        self.text_id = None # Id for text item

        self.create_image(self.filename)
        self.update_bbox()
        self.create_text() # New method call to create text
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

def update(self):
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_text() # New method call to update text
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def create_text(self): # New method to create text
    self.text_id = self.canvas.create_text(self.x1, self.y1-30,
                                           text=self.text, fill="black",
                                           font='Helvetica 10 bold',
                                           angle=self.angle, tags="text")

def update_text(self): # New Method to update text
    self.canvas.coords(self.text_id, self.x1, self.y1-30)

```

Comp_Lib/inductor.py

```

from pathlib import Path

from Comp_Lib.component import Component


class Inductor(Component):
    """Model for lumped element inductor"""
    def __init__(self, canvas, x1, y1, inductance): # Added inductance
argument
        super().__init__(canvas, x1, y1)
        self.inductance = inductance # Added inductance attribute

```

```

        self.filename = Path(__file__).parent /
"../images/lumped/inductor_60x30.png"

        self.in1, self.out = None, None
        self.conn_list = []

        self.text = 'L=' + str(inductance) + 'nH' # Create inductor text using
inductance
        self.text_id = None # Id for text item

        self.create_image(self.filename)
        self.update_bbox()
        self.create_text() # New method call to create text
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_text() # New method call to update text
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_text(self): # New method to create text
        self.text_id = self.canvas.create_text(self.x1, self.y1-30,
                                              text=self.text, fill="black",
                                              font='Helvetica 10 bold',
                                              angle=self.angle, tags="text")

    def update_text(self): # New method to update text
        self.canvas.coords(self.text_id, self.x1, self.y1-30)

```

Comp_Lib/capacitor.py

```

from pathlib import Path

from Comp_Lib.component import Component

class Capacitor(Component):

```

```

"""Model for lumped element capacitor"""
def __init__(self, canvas, x1, y1, capacitance): # Added capacitance argument
    super().__init__(canvas, x1, y1)
    self.capacitance = capacitance # Added capacitance attribute
    self.filename = Path(__file__).parent /
"../images/lumped/capacitor_60x30.png"

    self.in1, self.out = None, None
    self.conn_list = []

    self.text = 'C=' + str(capacitance) + 'pF' # Create capacitor text
using capacitance
    self.text_id = None # Id for text item

    self.create_image(self.filename)
    self.update_bbox()
    self.create_text() # New method call to create text
    self.create_selector()
    self.create_connectors()
    self.set_connector_visibility()

def update(self):
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_text() # New method call to update text
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def create_text(self): # New method to create text
    self.text_id = self.canvas.create_text(self.x1, self.y1-30,
                                           text=self.text, fill="black",
                                           font='Helvetica 10 bold',
                                           angle=self.angle, tags="text")

def update_text(self): # New method to update text
    self.canvas.coords(self.text_id, self.x1, self.y1-30)

```

UI_Lib/left_button_frame.py

```

    . . .

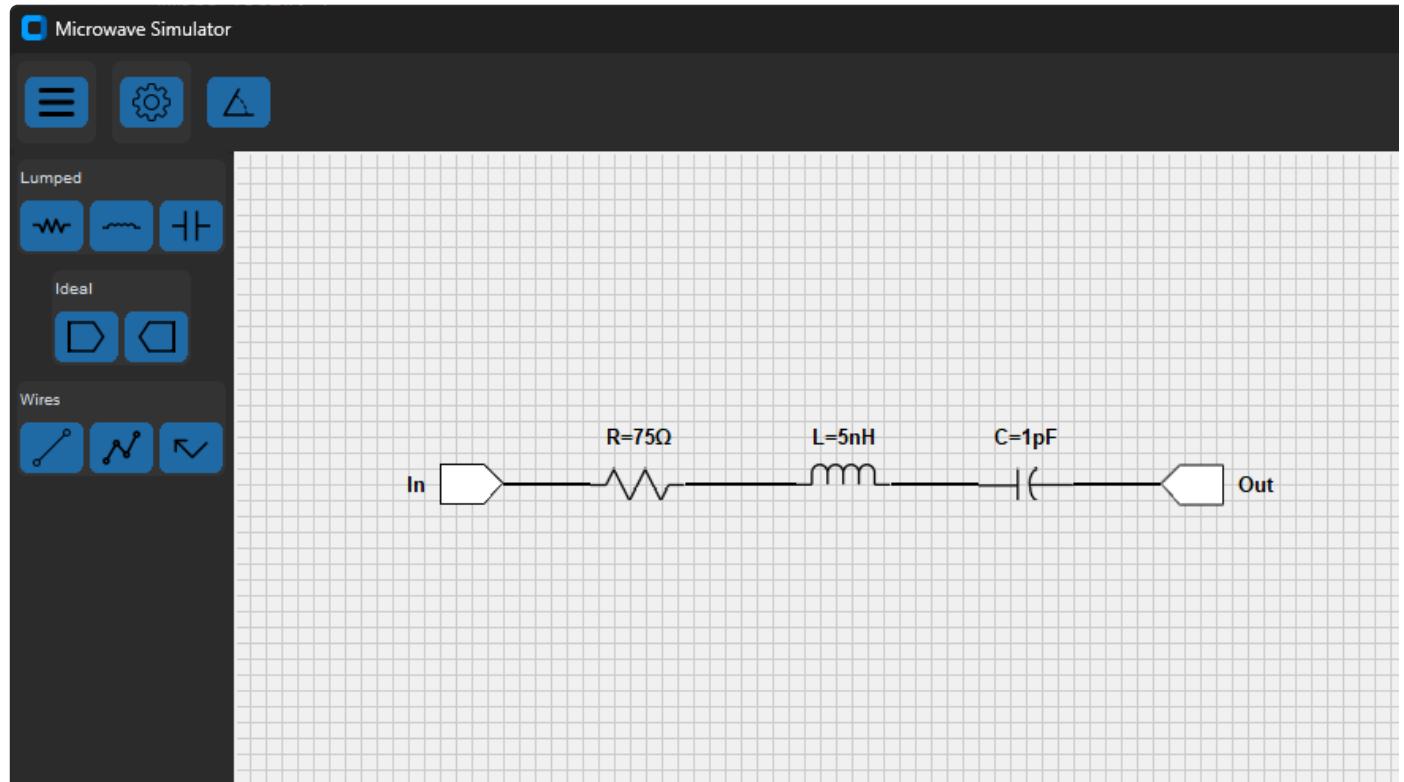
def create_events(self, name):
    gate = None
    if name == "resistor":
        gate = Resistor(self.canvas, 100, 100, 75) # Init resistor with
resistance
    elif name == "inductor":
        gate = Inductor(self.canvas, 100, 100, 5) # Init inductor with
inductance
    elif name == "capacitor":
        gate = Capacitor(self.canvas, 100, 100, 1) # Init capacitor with
capacitance
    self.canvas.comp_list.append(gate)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

    . . .

```

File Save & Load

Screen after file save & load



UI_Lib/file_menu_frame.py

```
import customtkinter as ctk
from tkinter import filedialog as fd
from pathlib import Path
import json
from PIL import Image

from Comp_Lib import Resistor, Inductor, Capacitor
from Comp_Lib import Inport, Outport, Connection
from Wire_Lib import StraightWire, SegmentWire, ElbowWire


class MyEncoder(json.JSONEncoder):
    def default(self, o):
        if hasattr(o, "reprJson"):
            return o.reprJson()
        else:
            return super().default(o)

class JSONDCoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self, object_hook=JSONDCoder.from_dict)

    @staticmethod
    def from_dict(_d):
        return _d


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.obj_type_dict = {'resistor': Resistor,
                             'inductor': Inductor,
                             'capacitor': Capacitor,
                             'inport': Inport,
                             'outport': Outport,
                             'straight': StraightWire,
                             'segment': SegmentWire,
                             'elbow': ElbowWire}

        self.menu_on = False
```

```

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
command=self.new_diagram)
        new_btn.pack(pady=5)

        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
command=window.destroy)
        exit_btn.pack(pady=5)

        my_image = ctk.CTkImage(light_image=Image.open
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                                dark_image=Image.open
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                                size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def new_diagram(self):
        self.canvas.delete("all")
        self.canvas.comp_list = []

    def load_diagram(self):
        try:
            filetypes = (('json files', '*.json'), ('All files', '*.*'))
            f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
            with open(f) as file:
                d = json.load(file)
                # print(d)

                for item in d:
                    if item['type'] == "straight":
                        wire = StraightWire(self.canvas, x1=item['x1'],

```

```

y1=item['y1'], x2=item['x2'], y2=item['y2'])
        self.canvas.comp_list.append(wire)
    elif item['type'] == "segment":
        wire = SegmentWire(self.canvas, x1=item['x1'],
y1=item['y1'], x2=item['x2'], y2=item['y2'])
        self.canvas.comp_list.append(wire)
    elif item['type'] == "elbow":
        wire = ElbowWire(self.canvas, x1=item['x1'],
y1=item['y1'], x2=item['x2'], y2=item['y2'])
        self.canvas.comp_list.append(wire)
    elif item['type'] == "resistor":
        resistor = Resistor(self.canvas, x1=item['x1'],
y1=item['y1'], resistance=item['resistance'])
        resistor.angle = item['angle']
        self.canvas.comp_list.append(resistor)
    elif item['type'] == "inductor":
        inductor = Inductor(self.canvas, x1=item['x1'],
y1=item['y1'], inductance=item['inductance'])
        inductor.angle = item['angle']
        self.canvas.comp_list.append(inductor)
    elif item['type'] == "capacitor":
        capacitor = Capacitor(self.canvas, x1=item['x1'],
y1=item['y1'],
                                capacitance=item['capacitance'])
        capacitor.angle = item['angle']
        self.canvas.comp_list.append(capacitor)
    elif item['type'] == "inport":
        inport = Inport(self.canvas, x1=item['x1'],
y1=item['y1'])
        inport.angle = item['angle']
        self.canvas.comp_list.append(inport)
    elif item['type'] == "outport":
        outport = Outport(self.canvas, x1=item['x1'],
y1=item['y1'])
        outport.angle = item['angle']
        self.canvas.comp_list.append(outport)

# Add connections
for item in d:
    if item['type'] == "resistor":
        wire_list = item['wire_list']
        for wire_item in wire_list:
            x1 = wire_item['wire_obj']['x1']
            y1 = wire_item['wire_obj']['y1']
            x2 = wire_item['wire_obj']['x2']
            y2 = wire_item['wire_obj']['y2']

```

```

        # Test to see if wire obj matches wire coordinates
        if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
            conn = Connection(resistor, wire,
wire_item['wire_end'])
            resistor.wire_list.append(conn)

    elif item['type'] == "inductor":
        wire_list = item['wire_list']
        for wire_item in wire_list:
            x1 = wire_item['wire_obj']['x1']
            y1 = wire_item['wire_obj']['y1']
            x2 = wire_item['wire_obj']['x2']
            y2 = wire_item['wire_obj']['y2']
            # Test to see if wire obj matches wire coordinates
            if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
                conn = Connection(inductor, wire,
wire_item['wire_end'])
                inductor.wire_list.append(conn)

    elif item['type'] == "capacitor":
        wire_list = item['wire_list']
        for wire_item in wire_list:
            x1 = wire_item['wire_obj']['x1']
            y1 = wire_item['wire_obj']['y1']
            x2 = wire_item['wire_obj']['x2']
            y2 = wire_item['wire_obj']['y2']
            # Test to see if wire obj matches wire coordinates
            if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
                conn = Connection(capacitor, wire,
wire_item['wire_end'])
                capacitor.wire_list.append(conn)

    elif item['type'] == "inport":
        wire_list = item['wire_list']
        for wire_item in wire_list:
            x1 = wire_item['wire_obj']['x1']
            y1 = wire_item['wire_obj']['y1']
            x2 = wire_item['wire_obj']['x2']
            y2 = wire_item['wire_obj']['y2']
            # Test to see if wire obj matches wire coordinates
            if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
                conn = Connection(inport, wire,

```

```

        wire_item['wire_end'])
                import.wire_list.append(conn)

            elif item['type'] == "outport":
                wire_list = item['wire_list']
                for wire_item in wire_list:
                    x1 = wire_item['wire_obj']['x1']
                    y1 = wire_item['wire_obj']['y1']
                    x2 = wire_item['wire_obj']['x2']
                    y2 = wire_item['wire_obj']['y2']
                    # Test to see if wire obj matches wire coordinates
                    if x1 == wire.x1 and y1 == wire.y1 and x2 ==
wire.x2 and y2 == wire.y2:
                        conn = Connection(outport, wire,
wire_item['wire_end'])
                        outport.wire_list.append(conn)

        except FileNotFoundError:
            with open('untitled.canvas', 'w') as _file:
                pass
        self.canvas.comp_list = []

    def save_diagram(self):
        filetypes = ((json files, '*.json'), ('All files', '*.*'))
        f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
        with open(f, 'w') as file:
            file.write(json.dumps(self.canvas.comp_list, cls=MyEncoder,
indent=4))

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

```

Add `reprJSON()` to each component file for the JSON encoder
 Comp_Lib/import.py and outport.py

... . . .

```
def reprJson(self):
    return dict(type=self.type, x1=self.x1, y1=self.y1, angle=self.angle,
    wire_list=self.wire_list)
```

Comp_Lib/resistor.py

```
...
def reprJson(self):
    return dict(type=self.type, x1=self.x1, y1=self.y1, angle=self.angle,
    resistance=self.resistance,
    wire_list=self.wire_list)
```

Comp_Lib/inductor.py

```
...
def reprJson(self):
    return dict(type=self.type, x1=self.x1, y1=self.y1, angle=self.angle,
    inductance=self.inductance,
    wire_list=self.wire_list)
```

Comp_Lib/capacitor.py

```
...
def reprJson(self):
    return dict(type=self.type, x1=self.x1, y1=self.y1, angle=self.angle,
    capacitance=self.capacitance,
    wire_list=self.wire_list)
```

Wire_lib/straight_wire.py & segment_wire.py & elbow_wire.py

```
    . . .
    def reprJson(self):
        return dict(type=self.type, x1=self.x1, y1=self.y1, x2=self.x2,
y2=self.y2)
```

```
rlc.json - file saved
```

```
[
{
    "type": "inport",
    "x1": 150,
    "y1": 210,
    "angle": 0,
    "wire_list": [
        {
            "type": "connection",
            "wire_obj": {
                "type": "straight",
                "x1": 170.0,
                "y1": 210.0,
                "x2": 225.0,
                "y2": 210.0
            },
            "wire_end": "begin"
        }
    ]
},
{
    "type": "resistor",
    "x1": 255,
    "y1": 210,
    "angle": 0,
    "resistance": 75,
    "wire_list": [
        {
            "type": "connection",
            "wire_obj": {
                "type": "straight",
                "x1": 170.0,
                "y1": 210.0,
                "x2": 225.0,
                "y2": 210.0
            }
        }
    ]
}
```

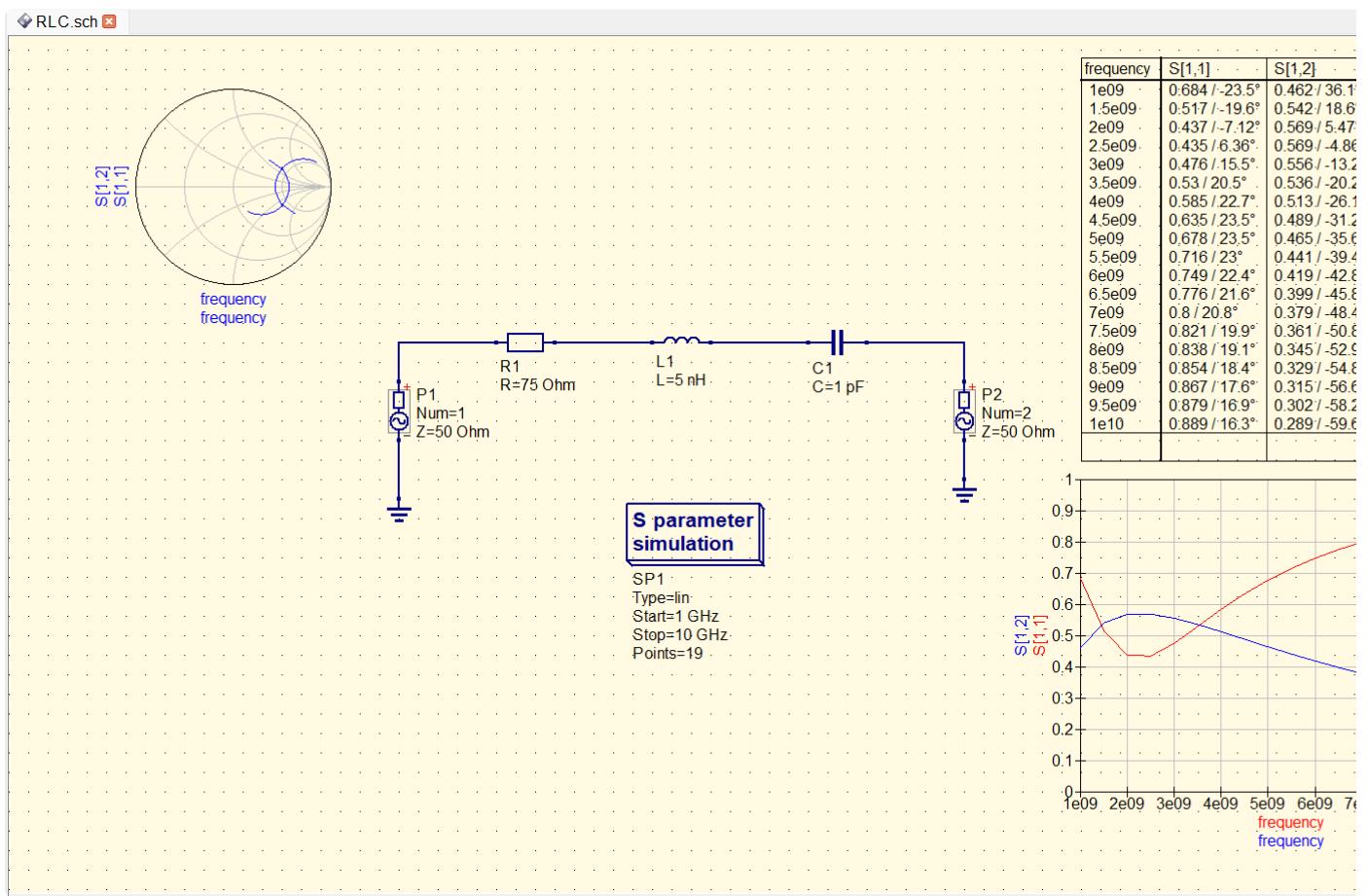
```
        },
        "wire_end": "end"
    },
    {
        "type": "connection",
        "wire_obj": {
            "type": "straight",
            "x1": 285.0,
            "y1": 210.0,
            "x2": 355.0,
            "y2": 210.0
        },
        "wire_end": "begin"
    }
]
},
{
    "type": "inductor",
    "x1": 385,
    "y1": 210,
    "angle": 0,
    "inductance": 5,
    "wire_list": [
        {
            "type": "connection",
            "wire_obj": {
                "type": "straight",
                "x1": 285.0,
                "y1": 210.0,
                "x2": 355.0,
                "y2": 210.0
            },
            "wire_end": "end"
        },
        {
            "type": "connection",
            "wire_obj": {
                "type": "straight",
                "x1": 415.0,
                "y1": 210.0,
                "x2": 470.0,
                "y2": 210.0
            },
            "wire_end": "begin"
        }
    ]
}
```

```
},
{
  "type": "capacitor",
  "x1": 500,
  "y1": 210,
  "angle": 0,
  "capacitance": 1,
  "wire_list": [
    {
      "type": "connection",
      "wire_obj": {
        "type": "straight",
        "x1": 415.0,
        "y1": 210.0,
        "x2": 470.0,
        "y2": 210.0
      },
      "wire_end": "end"
    },
    {
      "type": "connection",
      "wire_obj": {
        "type": "straight",
        "x1": 530.0,
        "y1": 210.0,
        "x2": 585.0,
        "y2": 210.0
      },
      "wire_end": "begin"
    }
  ]
},
{
  "type": "outport",
  "x1": 605,
  "y1": 210,
  "angle": 0,
  "wire_list": [
    {
      "type": "connection",
      "wire_obj": {
        "type": "straight",
        "x1": 530.0,
        "y1": 210.0,
        "x2": 585.0,
        "y2": 210.0
      }
    }
  ]
}
```

```
        },
        "wire_end": "end"
    }
]
},
{
    "type": "straight",
    "x1": 170.0,
    "y1": 210.0,
    "x2": 225.0,
    "y2": 210.0
},
{
    "type": "straight",
    "x1": 285.0,
    "y1": 210.0,
    "x2": 355.0,
    "y2": 210.0
},
{
    "type": "straight",
    "x1": 415.0,
    "y1": 210.0,
    "x2": 470.0,
    "y2": 210.0
},
{
    "type": "straight",
    "x1": 530.0,
    "y1": 210.0,
    "x2": 585.0,
    "y2": 210.0
}
]
```

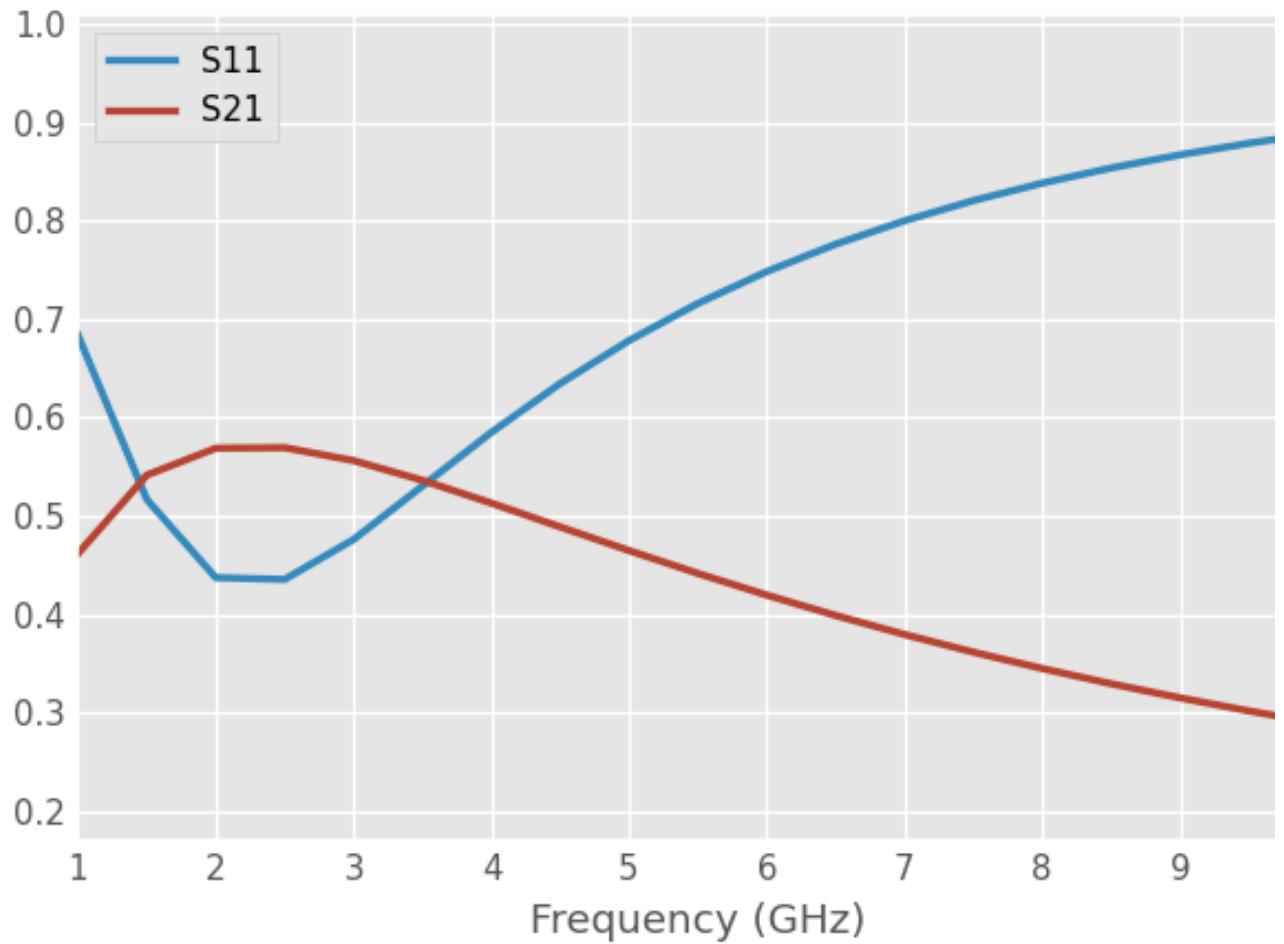
RLC Circuit Analysis

QUCS Analysis



Scikit-RF Analysis

Figure 1



Console output

```
D:\EETools\MicrowaveSimulator\venv\Scripts\python.exe
D:\EETools\MicrowaveSimulator\Sandbox\RLC_example.py
[[[0.68368911 0.46154922]
 [0.46154922 0.68368911]]

 [[0.51666132 0.54150201]
 [0.54150201 0.51666132]]

 [[0.43712731 0.56883028]
 [0.56883028 0.43712731]]]
```

```
[[0.43535197 0.56937463]  
[0.56937463 0.43535197]]
```

```
[[0.47595789 0.55622445]  
[0.55622445 0.47595789]]
```

```
[[0.53033858 0.5361869 ]  
[0.5361869 0.53033858]]
```

```
[[0.58489041 0.51299248]  
[0.51299248 0.58489041]]
```

```
[[0.63466307 0.48875466]  
[0.48875466 0.63466307]]
```

```
[[0.67836989 0.46467808]  
[0.46467808 0.67836989]]
```

```
[[0.71612899 0.4414337 ]  
[0.4414337 0.71612899]]
```

```
[[0.7485435 0.41937221]  
[0.41937221 0.7485435 ]]
```

```
[[0.77633296 0.39865129]  
[0.39865129 0.77633296]]
```

```
[[0.80018956 0.37931342]  
[0.37931342 0.80018956]]
```

```
[[0.82072802 0.36133393]  
[0.36133393 0.82072802]]
```

```
[[0.83847445 0.34465089]  
[0.34465089 0.83847445]]
```

```
[[0.85387096 0.32918346]  
[0.32918346 0.85387096]]
```

```
[[0.86728573 0.31484312]
```

```
[0.31484312 0.86728573]]
```

```
[[0.87902421 0.30154034]  
[0.30154034 0.87902421]]
```

```
[[0.88933967 0.28918848]  
[0.28918848 0.88933967]]]
```

```
Process finished with exit code 0
```

Sandbox/RLC_example.py

```
import skrf as rf  
import matplotlib.pyplot as plt  
rf.stylely()  
  
freq = rf.Frequency(start=1.0, stop=10.0, unit='GHz', npoints=19)  
tl_media = rf.DefinedGammaZ0(freq, z0=50)  
R1 = rf.Circuit.SeriesImpedance(frequency=freq, name='R1', z0=50,  
                                Z=75.0)  
L1 = rf.Circuit.SeriesImpedance(frequency=freq, name='L1', z0=50,  
                                Z=1j*freq.w*5e-9)  
C1 = rf.Circuit.SeriesImpedance(frequency=freq, name='C1', z0=50,  
                                Z=1/(1j*freq.w*1e-12))  
  
port1 = rf.Circuit.Port(freq, name='port1', z0=50)  
port2 = rf.Circuit.Port(freq, name='port2', z0=50)  
  
cnx = [  
    [(port1, 0), (R1, 0)], # port1 output to R1 input  
    [(R1, 1), (L1, 0)], # R1 output to L1 input  
    [(L1, 1), (C1, 0)], # L1 output to C1 input  
    [(C1, 1), (port2, 0)] # C1 output to port2 input  
]  
ckt = rf.Circuit(cnx)  
ntw = ckt.network  
  
print(ntw.s_mag)  
  
ntw.plot_s_mag(m=0, n=0, lw=2, logx=False)  
ntw.plot_s_mag(m=1, n=0, lw=2, logx=False)
```

```
plt.show()
```

Basically, we need to convert the circuit diagram to a circuit list with connections and run the analysis.

Component List Analysis

Sandbox/RLC_component_list_processor.py

```
import skrf as rf
import matplotlib.pyplot as plt
rf.stylely()

class Resistor:
    def __init__(self, x, y, resistance):
        self.type = 'resistor'

class Inductor:
    def __init__(self, x, y, inductance):
        self.type = 'inductor'

class Capacitor:
    def __init__(self, x, y, capacitance):
        self.type = 'capacitor'

class Inport:
    def __init__(self, x, y):
        self.type = 'inport'

class Outport:
    def __init__(self, x, y):
        self.type = 'outport'

class StraightWire:
```

```

def __init__(self, x1, y1, x2, y2):
    self.type = 'wire'

# Create component list
def create_comp_list():
    # Components
    comp_list = []
    Res = Resistor(100, 100, 75)
    comp_list.append(Res)
    Ind = Inductor(100, 100, 5)
    comp_list.append(Ind)
    Cap = Capacitor(100, 100, 1)
    comp_list.append(Cap)
    inp = Import(100, 100)
    comp_list.append(inp)
    outp = Outport(100, 100)
    comp_list.append(outp)

    # Wires
    wire1 = StraightWire(100, 100, 200, 200)
    wire2 = StraightWire(100, 100, 200, 200)
    wire3 = StraightWire(100, 100, 200, 200)
    wire4 = StraightWire(100, 100, 200, 200)
    comp_list.append(wire1)
    comp_list.append(wire2)
    comp_list.append(wire3)
    comp_list.append(wire4)

    # Create wire list
    wire_list = [
        [(inp, 'out'), (Res, 'in1')],
        [(Res, 'out'), (Ind, 'in1')],
        [(Ind, 'out'), (Cap, 'in1')],
        [(Cap, 'out'), (outp, 'in1')]
    ]
    return comp_list, wire_list

def analyze(component_list, wire_list):
    cnx = []
    R1, L1, C1, port1, port2 = None, None, None, None, None
    freq = rf.Frequency(start=1.0, stop=10.0, unit='GHz', npoints=19)

    # Convert graphical components to microwave components

```

```

    for comp in component_list:
        if isinstance(comp, Resistor):
            R1 = rf.Circuit.SeriesImpedance(frequency=freq, name='R1', z0=50,
Z=75.0)
        elif isinstance(comp, Inductor):
            L1 = rf.Circuit.SeriesImpedance(frequency=freq, name='L1', z0=50,
Z=1j * freq.w * 5e-9)
        elif isinstance(comp, Capacitor):
            C1 = rf.Circuit.SeriesImpedance(frequency=freq, name='C1', z0=50,
Z=1 / (1j * freq.w * 1e-12))
        elif isinstance(comp, Inport):
            port1 = rf.Circuit.Port(freq, name='port1', z0=50)
        elif isinstance(comp, Outport):
            port2 = rf.Circuit.Port(freq, name='port2', z0=50)

    comp_dict = {
        'resistor': R1,
        'inductor': L1,
        'capacitor': C1,
        'inport': port1,
        'outport': port2,
        'wire': None
    }

    wire_end_dict = {
        'in1': 0,
        'out': 1
    }

# Convert wire list to cnx list
print(wire_list)
for wire in wire_list:
    # Parse wire list
    comp1 = wire[0][0].type
    if comp1 == "inport":
        end1 = 'in1' # Hack for Scikit-RF input port which labels the
output port as 0, I labeled them 'out'
    else:
        end1 = wire[0][1]
    comp2 = wire[1][0].type
    end2 = wire[1][1]

    # Convert to cnx list
    a_cnx = [(comp_dict[comp1], wire_end_dict[end1]), (comp_dict[comp2],
wire_end_dict[end2])]
    cnx.append(a_cnx)

```

```
# cnx = [
#     [(port1, 0), (R1, 0)], # port1 output to R1 input
#     [(R1, 1), (L1, 0)], # R1 output to L1 input
#     [(L1, 1), (C1, 0)], # L1 output to C1 input
#     [(C1, 1), (port2, 0)] # C1 output to port2 input
# ]

ckt = rf.Circuit(cnx)
ntw = ckt.network

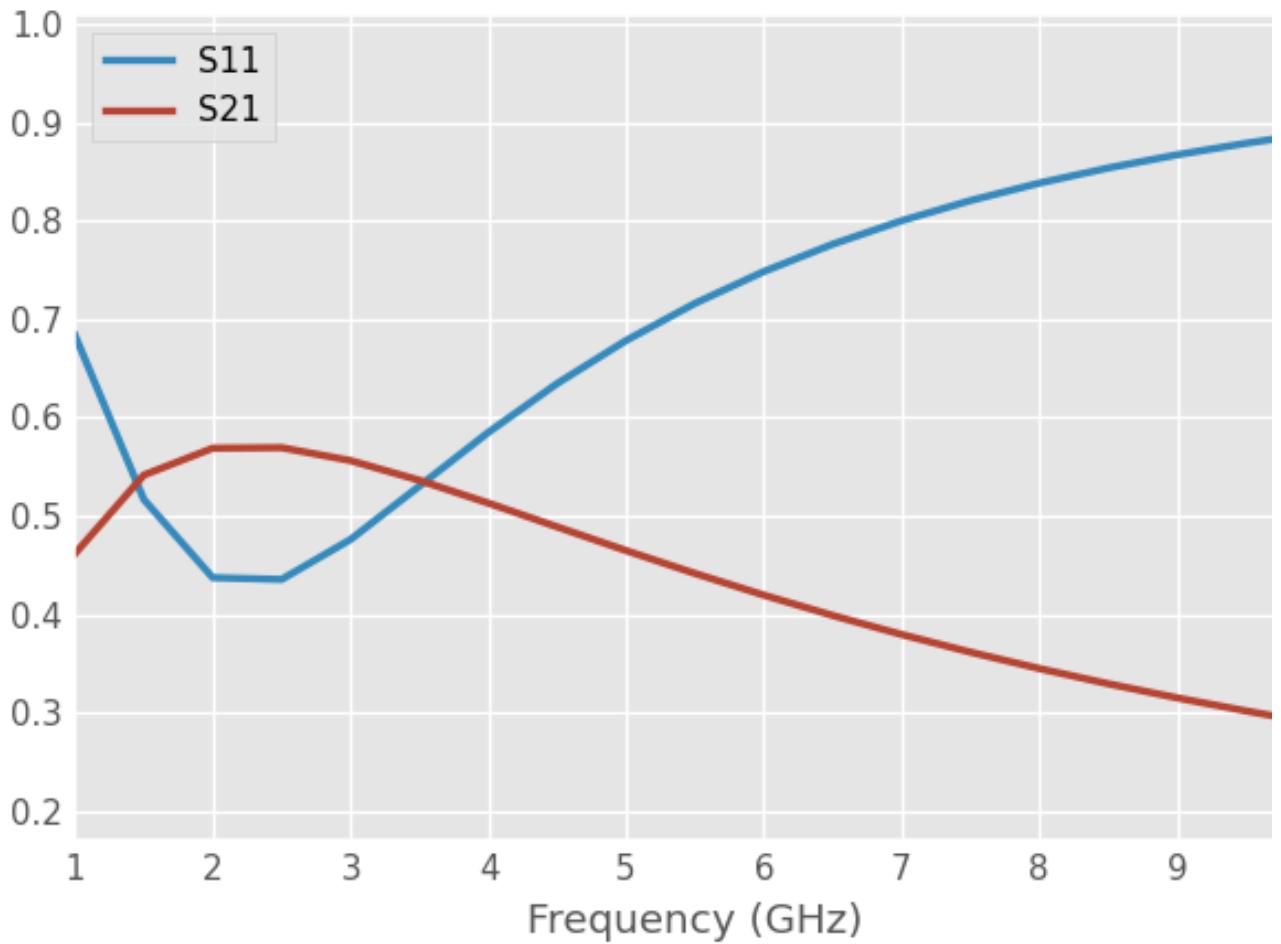
print(ntw.s_mag)

ntw.plot_s_mag(m=0, n=0, lw=2, logx=False)
ntw.plot_s_mag(m=1, n=0, lw=2, logx=False)
plt.show()

# Analyze the circuit
a_comp_list, a_wire_list = create_comp_list()
analyze(a_comp_list, a_wire_list)
```



Figure 1



The key here is to use the component type attribute (used for file save/load also) to create a dictionary that maps graphical component classes to microwave components. I also used a dictionary to map connection names ('in1', 'out') to 0, 1 respectively, which is the notation used for Scikit-RF component ports.

Analysis

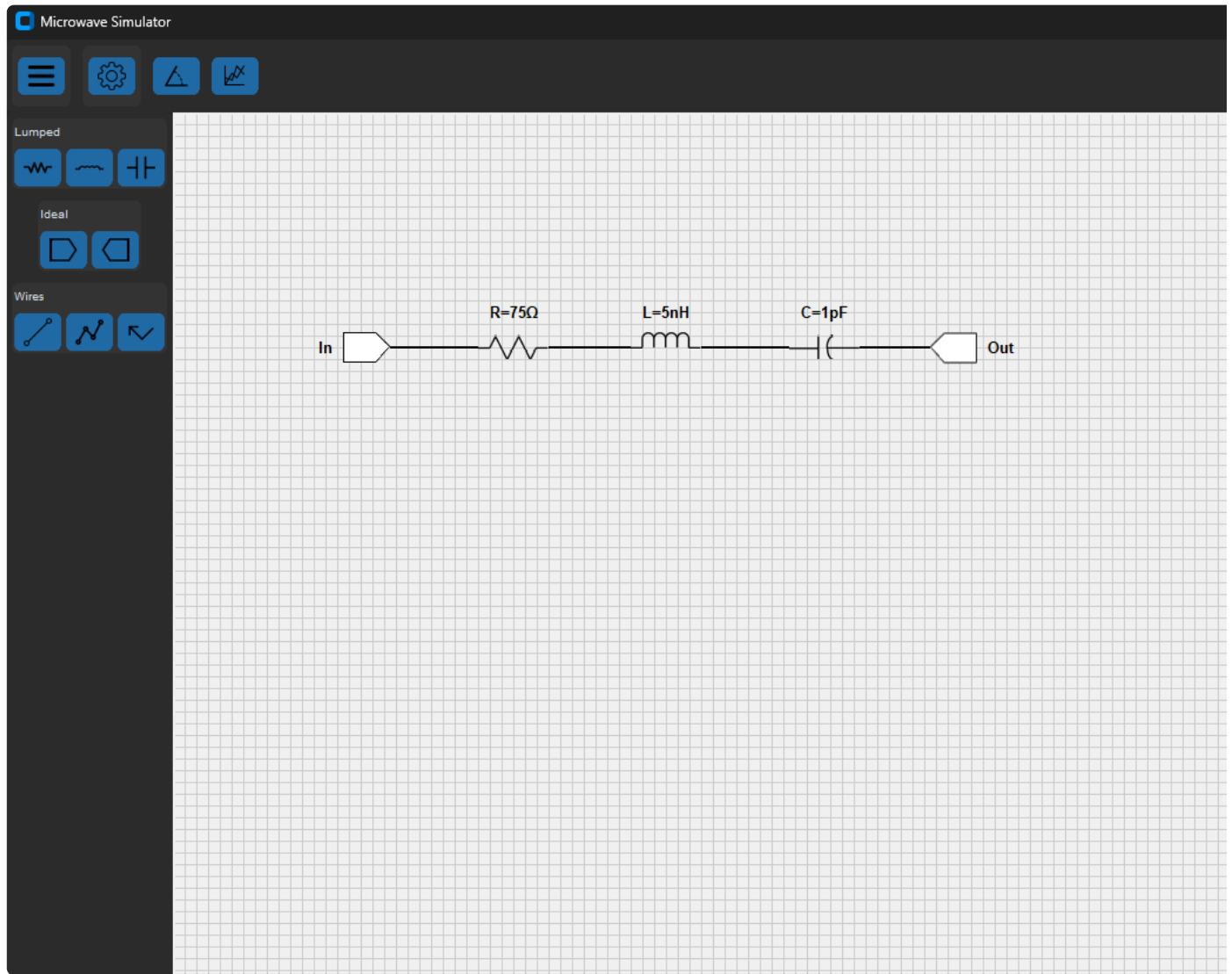
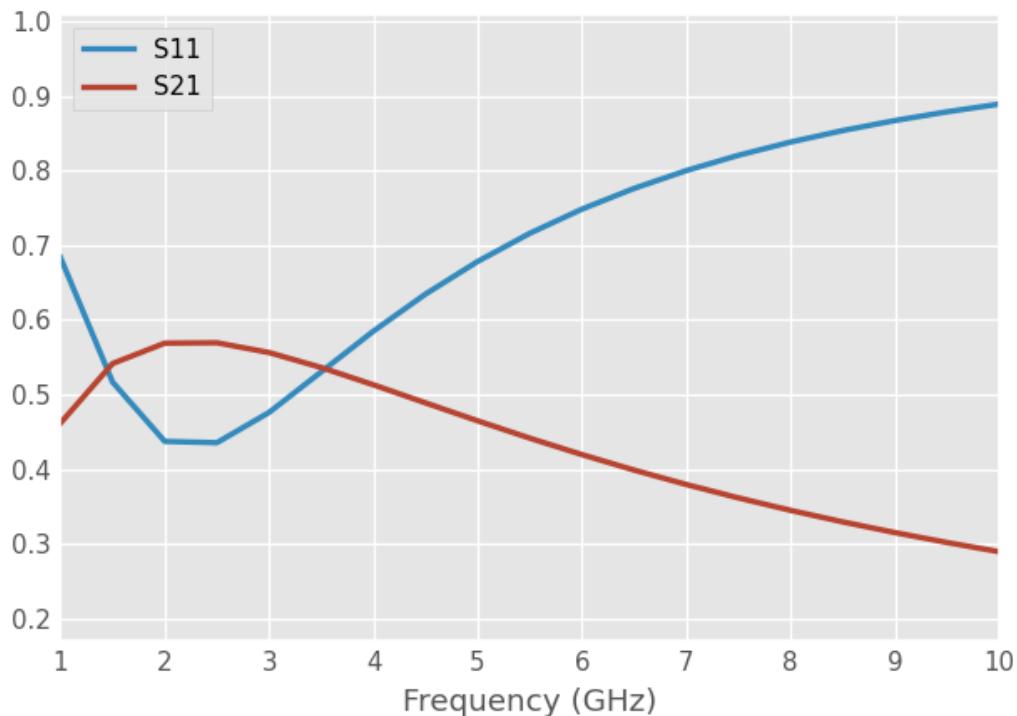


Figure 1



Console output

```
[[[0.68368911 0.46154922]
```

```
[0.46154922 0.68368911]]
```

```
[[0.51666132 0.54150201]
```

```
[0.54150201 0.51666132]]
```

```
[[0.43712731 0.56883028]
```

```
[0.56883028 0.43712731]]
```

```
[[0.43535197 0.56937463]
```

```
[0.56937463 0.43535197]]
```

```
[[0.47595789 0.55622445]
```

```
[0.55622445 0.47595789]]
```

```
[[0.53033858 0.5361869 ]
```

```
[0.5361869 0.53033858]]
```

[[0.58489041 0.51299248]
[0.51299248 0.58489041]]

[[0.63466307 0.48875466]
[0.48875466 0.63466307]]

[[0.67836989 0.46467808]
[0.46467808 0.67836989]]

[[0.71612899 0.4414337]
[0.4414337 0.71612899]]

[[0.7485435 0.41937221]
[0.41937221 0.7485435]]

[[0.77633296 0.39865129]
[0.39865129 0.77633296]]

[[0.80018956 0.37931342]
[0.37931342 0.80018956]]

[[0.82072802 0.36133393]
[0.36133393 0.82072802]]

[[0.83847445 0.34465089]
[0.34465089 0.83847445]]

[[0.85387096 0.32918346]
[0.32918346 0.85387096]]

[[0.86728573 0.31484312]
[0.31484312 0.86728573]]

[[0.87902421 0.30154034]
[0.30154034 0.87902421]]

[[0.88933967 0.28918848]
[0.28918848 0.88933967]]]

UI.Lib/top_frame.py

```
import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

import skrf as rf # Added new import
import matplotlib.pyplot as plt # Added new import
rf.stylely() # Added

from UI.Lib.file_menu_frame import FileMenuFrame
from UI.Lib.settings_frame import SettingsFrame
from UI.Lib.help_frame import HelpFrame

from Comp.Lib import Resistor, Inductor, Capacitor, Import, Outport # Added
new import

class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add Top Frame widget here
        file_frame = FileMenuFrame(self.parent, self, self.canvas)
        file_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        settings_frame = SettingsFrame(self.parent, self, self.canvas)
        settings_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        help_frame = HelpFrame(self.parent, self, self.canvas)
        help_frame.pack(side=ctk.RIGHT, padx=5, pady=5)

        a_image = ctk.CTkImage(light_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
                               dark_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
                               size=(24, 24))
        self.button_id = ctk.CTkButton(self, text="", image=a_image, width=30,
command=self.rotate_comp)
        self.button_id.pack(side=ctk.LEFT, padx=5, pady=5)
        ToolTip(self.button_id, msg="Rotate selected component")
```

```

        a_image = ctk.CTkImage(light_image=Image.open(Path(__file__).parent /
"../icons/analyze.png"),
                               dark_image=Image.open(Path(__file__).parent /
"../icons/analyze.png"),
                               size=(24, 24))
    self.button_id = ctk.CTkButton(self, text="", image=a_image, width=30,
command=self.analyze_circuit) # Call to analyze method
    self.button_id.pack(side=ctk.LEFT, padx=5, pady=5)
    ToolTip(self.button_id, msg="Analyze circuit")

def rotate_comp(self):
    self.parent.rotate_comp(_event=None)

def analyze_circuit(self): # New analyze method
    comp_list = self.canvas.comp_list
    wire_list = self.canvas.wire_list
    cnx = []

    res, ind, cap, inport, outport = None, None, None, None, None
    freq = rf.Frequency(start=1.0, stop=10.0, unit='GHz', npoints=19)

    # Convert graphical components to microwave components
    for comp in comp_list:
        if isinstance(comp, Resistor):
            res = rf.Circuit.SeriesImpedance(frequency=freq, name='res',
z0=50, Z=75.0)
        elif isinstance(comp, Inductor):
            ind = rf.Circuit.SeriesImpedance(frequency=freq, name='ind',
z0=50, Z=1j * freq.w * 5e-9)
        elif isinstance(comp, Capacitor):
            cap = rf.Circuit.SeriesImpedance(frequency=freq, name='cap',
z0=50, Z=1 / (1j * freq.w * 1e-12))
        elif isinstance(comp, Import):
            inport = rf.Circuit.Port(freq, name='inport', z0=50)
        elif isinstance(comp, Outport):
            outport = rf.Circuit.Port(freq, name='outport', z0=50)

    comp_dict = {
        'resistor': res,
        'inductor': ind,
        'capacitor': cap,
        'inport': inport,
        'outport': outport,
        'wire': None
    }

```

```

    wire_end_dict = {
        'in1': 0,
        'out': 1
    }

    # Convert wire list to cnx list
    print(wire_list)
    for wire in wire_list:
        # Parse wire list
        comp1 = wire[0][0].type
        if comp1 == "inport":
            end1 = 'in1' # Hack for Scikit-RF input port which labels the
output port as 0, I labeled them 'out'
        else:
            end1 = wire[0][1]
        comp2 = wire[1][0].type
        end2 = wire[1][1]

        # Convert to cnx list
        a_cnx = [(comp_dict[comp1], wire_end_dict[end1]),
(comp_dict[comp2], wire_end_dict[end2])]
        cnx.append(a_cnx)

    ckt = rf.Circuit(cnx)
    ntw = ckt.network

    print(ntw.s_mag) # Print S-parameters to console

    ntw.plot_s_mag(m=0, n=0, lw=2, logx=False)
    ntw.plot_s_mag(m=1, n=0, lw=2, logx=False)
    plt.show() # Display S-parameter plot

```

UI_Lib/canvas.py

```

import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid


class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)

```

```

    self.led_color = "red"
    self.led_size = "large"
    self.grid = Grid(self, 10)
    self.mouse = Mouse(self)
    self.comp_list = []
    self.wire_list = [] # Added new wire list

    self.mouse.move_mouse_bind_events()

    . . .

```

Wire_Lib/wire.py

```

from Wire_Lib.wire_selector import WireSelector

class Wire:
    """Base class for wire classes"""
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        self.fill_color = "black"
        self.border_width = 2
        self.line_direction = "horizontal"

        self.id = None
        self.is_selected = False
        self.selector = None
        self.width = 2
        self.bbox = None

    # Connections for wire list
    self.in_cnx = None # Added
    self.out_cnx = None # Added
    self.cnx = [] # Added

    self.sel1, self.sel2 = None, None

```

```

    . . .

def create_wire_list_cnx(self, comp, end): # Added new method - called
from Mouse Class
    if end == 'out':
        self.out_cnx = (comp, end)
    elif end == 'in1':
        self.in_cnx = (comp, end)
    if self.in_cnx and self.out_cnx:
        self.cnx = [self.out_cnx, self.in_cnx]
        self.canvas.wire_list.append(self.cnx)

```

UI_Lib/mouse.py

```

    . . .

def select_connector(self, wire_obj, wire_end, x, y): # Added method to
see if line end hits a gate connector
    for comp in self.canvas.comp_list:
        if not (isinstance(comp, StraightWire) or isinstance(comp,
SegmentWire) or
                isinstance(self.selected_comp, ElbowWire)):
            conn = comp.check_connector_hit(x, y)
            if conn:
                if wire_end == "begin":
                    wire_obj.x1, wire_obj.y1 = conn.x, conn.y
                elif wire_end == "end":
                    wire_obj.x2, wire_obj.y2 = conn.x, conn.y
                    a_conn = Connection(conn, self.current_wire_obj, wire_end)
                    wire_obj.create_wire_list_cnx(comp, conn.name) # Added
method call to creeate a wire connection (cnx)
                    comp.wire_list.append(a_conn)
                    self.canvas.redraw()

```

Mini Microwave Simulator

New

Save

Open

Analyze

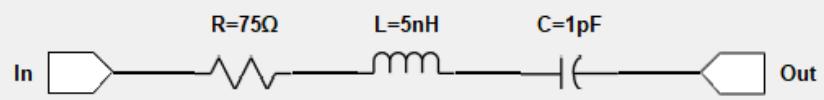
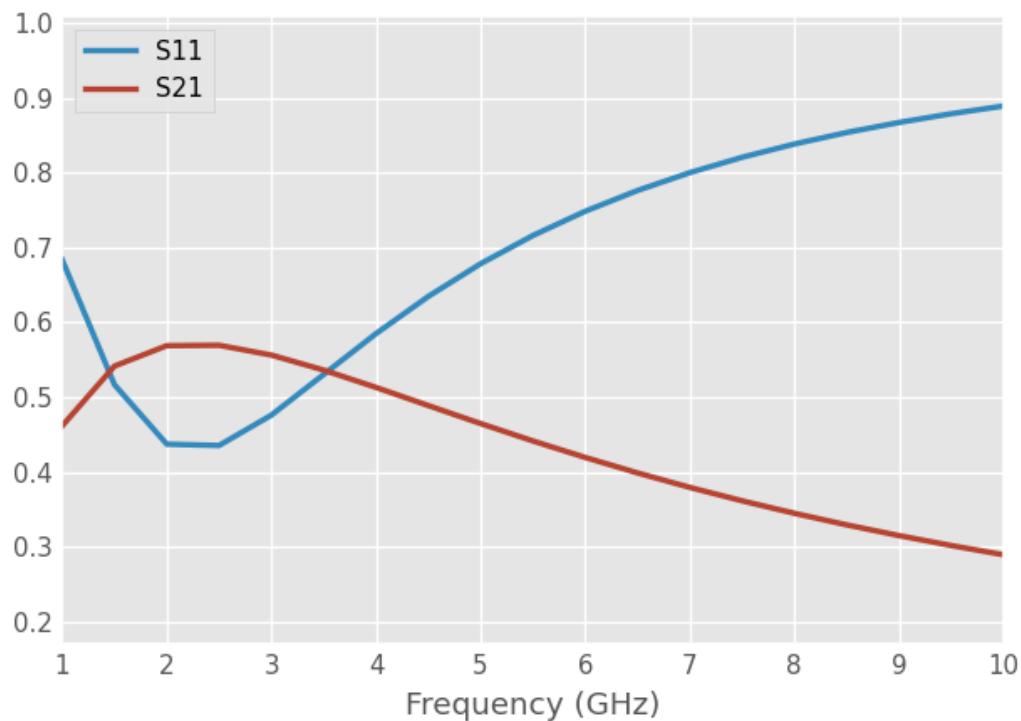
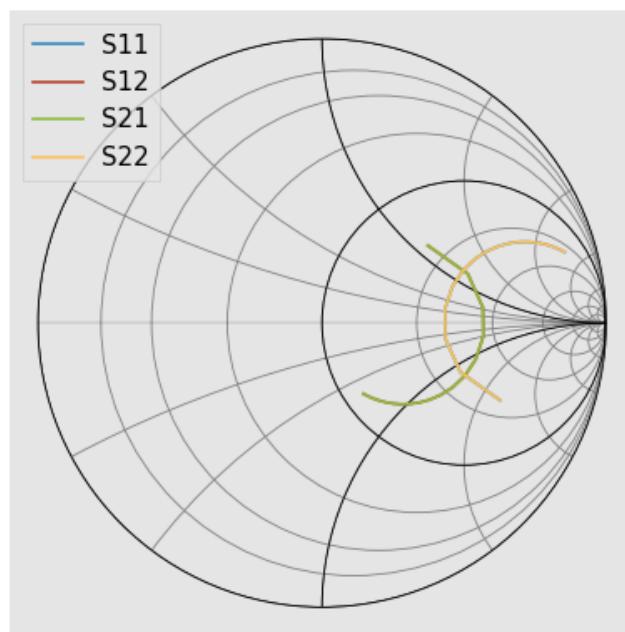


Figure 1



Home | Back | Forward | Crosshair | Search | Zoom | Print

Figure 1



Home | Back | Forward | Crosshair | Search | Zoom | Print

Console output

```
D:\EETools\MicrowaveSimulator\venv\Scripts\python.exe
D:\EETools\MicrowaveSimulator\Sandbox\mini_microwave_simulator.py
[[[0.68368911 0.46154922]
 [0.46154922 0.68368911]]

 [[0.51666132 0.54150201]
 [0.54150201 0.51666132]]

 [[0.43712731 0.56883028]
 [0.56883028 0.43712731]]

 [[0.43535197 0.56937463]
 [0.56937463 0.43535197]]

 [[0.47595789 0.55622445]
 [0.55622445 0.47595789]]

 [[0.53033858 0.5361869 ]
 [0.5361869  0.53033858]]

 [[0.58489041 0.51299248]
 [0.51299248 0.58489041]]

 [[0.63466307 0.48875466]
 [0.48875466 0.63466307]]

 [[0.67836989 0.46467808]
 [0.46467808 0.67836989]]

 [[0.71612899 0.4414337 ]
 [0.4414337  0.71612899]]

 [[0.7485435  0.41937221]
 [0.41937221 0.7485435 ]]

 [[0.77633296 0.39865129]
 [0.39865129 0.77633296]]
```

```
[[0.80018956 0.37931342]  
[0.37931342 0.80018956]]
```

```
[[0.82072802 0.36133393]  
[0.36133393 0.82072802]]
```

```
[[0.83847445 0.34465089]  
[0.34465089 0.83847445]]
```

```
[[0.85387096 0.32918346]  
[0.32918346 0.85387096]]
```

```
[[0.86728573 0.31484312]  
[0.31484312 0.86728573]]
```

```
[[0.87902421 0.30154034]  
[0.30154034 0.87902421]]
```

```
[[0.88933967 0.28918848]  
[0.28918848 0.88933967]]]
```

Sandbox/mini_microwave_simulator.py

```
import customtkinter as ctk  
from Comp_Lib import Import, Outport, Resistor, Inductor, Capacitor  
from Wire_Lib import StraightWire  
  
import skrf as rf  
import matplotlib.pyplot as plt  
rf.stylely()  
  
import json  
from tkinter import filedialog as fd  
  
  
class Encoder(json.JSONEncoder):  
    def default(self, o):  
        if hasattr(o, "reprJson"):  
            return o.reprJson()  
        else:
```

```

        return super().default(o)

class Decoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self, object_hook=Decoder.from_dict)

    @staticmethod
    def from_dict(_d):
        return _d


class Connection:
    def __init__(self, comp_conn, wire_name, wire_end):
        self.type = "connection"
        self.connector_obj = comp_conn      # in1, out
        self.wire_obj = wire_name           # wire string name
        self.wire_end = wire_end            # "begin" or "end"

    def reprJson(self):
        return dict(type=self.type, connector_obj=self.connector_obj,
                    wire_obj=self.wire_obj, wire_end=self.wire_end)


class App(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100")  # w, h, x, y
        self.title("Mini Microwave Simulator")

        self.canvas = ctk.CTkCanvas(self)
        self.left_frame = ctk.CTkFrame(self)
        self.top_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add frame widgets here
        button = ctk.CTkButton(self.top_frame, text="New", command=self.new)
        button.pack(side=ctk.LEFT, padx=5, pady=5)

        button = ctk.CTkButton(self.top_frame, text="Save",
                               command=self.save_file)
        button.pack(side=ctk.LEFT, padx=5, pady=5)

```

```

        button = ctk.CTkButton(self.top_frame, text="Open",
command=self.open_file)
        button.pack(side=ctk.LEFT, padx=5, pady=5)

        button = ctk.CTkButton(self.top_frame, text="Analyze",
command=self.analyze)
        button.pack(side=ctk.LEFT, padx=5, pady=5)

# Add component list here
port1 = Import(self.canvas, 100, 100)
R1 = Resistor(self.canvas, 200, 100, 75)
L1 = Inductor(self.canvas, 300, 100, 5)
C1 = Capacitor(self.canvas, 400, 100, 1)
port2 = Outport(self.canvas, 500, 100)
wire1 = StraightWire(self.canvas, 120, 100, 170, 100)
wire2 = StraightWire(self.canvas, 230, 100, 270, 100)
wire3 = StraightWire(self.canvas, 330, 100, 370, 100)
wire4 = StraightWire(self.canvas, 430, 100, 480, 100)
self.comp_list = [port1, R1, L1, C1, port2, wire1, wire2, wire3, wire4]
self.comp_dict = {'comp_list': self.comp_list}

# Add wire connections here
port1.wire_list.append(Connection('out', 'wire1', 'begin'))
R1.wire_list.append(Connection('in1', 'wire1', 'end'))
R1.wire_list.append(Connection('out', 'wire2', 'begin'))
L1.wire_list.append(Connection('in1', 'wire2', 'end'))
L1.wire_list.append(Connection('out', 'wire3', 'begin'))
C1.wire_list.append(Connection('in1', 'wire3', 'end'))
C1.wire_list.append(Connection('out', 'wire4', 'begin'))
port2.wire_list.append(Connection('in1', 'wire4', 'end'))

# Add Scikit-RF connections here
conn1 = [('inport', 0), ('resistor', 0)]
conn2 = [('resistor', 1), ('inductor', 0)]
conn3 = [('inductor', 1), ('capacitor', 0)]
conn4 = [('capacitor', 1), ('outport', 0)]
self.conn_list = [conn1, conn2, conn3, conn4]
self.conn_dict = {'conn_list': self.conn_list}

self.circuit = [self.comp_dict, self.conn_dict]

def new(self):
    self.comp_list.clear()
    self.conn_list.clear()
    self.canvas.delete('all')

```

```

def save_file(self):
    filetypes = (('json files', '*.json'), ('All files', '*.*'))
    f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
    with open(f, 'w') as file:
        file.write(json.dumps(self.circuit, cls=Encoder, indent=4))

def open_file(self):
    try:
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
        with open(f) as file:
            d = json.load(file)
            self.convert_json_data(d)
    except FileNotFoundError:
        with open('untitled.canvas', 'w') as _file:
            pass

def convert_json_data(self, data):
    """Convert json data to a circuit object"""
    # Get circuit list from json data
    json_comp_list = data[0]['comp_list']
    for json_comp in json_comp_list:
        if json_comp['type'] == 'resistor':
            res = Resistor(self.canvas, int(json_comp['x1']),
int(json_comp['y1']), int(json_comp['resistance']))
            res.wire_list = json_comp['wire_list']
            self.comp_list.append(res)
        elif json_comp['type'] == 'inductor':
            ind = Inductor(self.canvas, int(json_comp['x1']),
int(json_comp['y1']), int(json_comp['inductance']))
            ind.wire_list = json_comp['wire_list']
            self.comp_list.append(ind)
        elif json_comp['type'] == 'capacitor':
            cap = Capacitor(self.canvas, int(json_comp['x1']),
int(json_comp['y1']), int(json_comp['capacitance']))
            cap.wire_list = json_comp['wire_list']
            self.comp_list.append(cap)
        elif json_comp['type'] == 'inport':
            port1 = Inport(self.canvas, int(json_comp['x1']),
int(json_comp['y1']))
            port1.wire_list = json_comp['wire_list']
            self.comp_list.append(port1)
        elif json_comp['type'] == 'outport':
            port2 = Outport(self.canvas, int(json_comp['x1']),
int(json_comp['y1']))
            port2.wire_list = json_comp['wire_list']

```

```

        self.comp_list.append(port2)
    elif json_comp['type'] == 'straight':
        wire = StraightWire(self.canvas, int(json_comp['x1']),
int(json_comp['y1']), int(json_comp['x2']),
int(json_comp['y2']))
        self.comp_list.append(wire)

    # Get connection list from json data
    self.conn_list = data[1]['conn_list']

def analyze(self):
    ctx = []
    res, ind, cap, inport, outport = None, None, None, None, None
    freq = rf.Frequency(start=1.0, stop=10.0, unit='GHz', npoints=19)

    # Convert graphical components to microwave components
    for comp in self.comp_list:
        if isinstance(comp, Resistor):
            res = rf.Circuit.SeriesImpedance(frequency=freq, name='res',
z0=50, Z=comp.resistance)
        elif isinstance(comp, Inductor):
            ind = rf.Circuit.SeriesImpedance(frequency=freq, name='ind',
z0=50,
Z=1j * freq.w *
comp.inductance * 1e-9)
        elif isinstance(comp, Capacitor):
            cap = rf.Circuit.SeriesImpedance(frequency=freq, name='cap',
z0=50,
Z=1 / (1j * freq.w *
comp.capacitance * 1e-12))
        elif isinstance(comp, Inport):
            inport = rf.Circuit.Port(freq, name='inport', z0=50)
        elif isinstance(comp, Outport):
            outport = rf.Circuit.Port(freq, name='outport', z0=50)

        comp_dict = {
            'resistor': res,
            'inductor': ind,
            'capacitor': cap,
            'inport': inport,
            'outport': outport,
            'wire': None
        }

        for conn in self.conn_list:
            ctx_conn = [(comp_dict[conn[0][0]], conn[0][1]), (comp_dict[conn[1]

```

```

[0]], conn[1][1]])
    ctx.append(ctx_conn)

ckt = rf.Circuit(ctx)
ntw = ckt.network

print(ntw.s_mag) # Print S-parameters to console

ntw.plot_s_mag(m=0, n=0, lw=2, logx=False)
ntw.plot_s_mag(m=1, n=0, lw=2, logx=False)
plt.show() # Display S-parameter plot

ntw.plot_s_smith()
plt.show()

if __name__ == "__main__":
    app = App()
    app.mainloop()

```

Sandbox/ RLC.json

```

[
{
    "comp_list": [
        {
            "type": "inport",
            "x1": 100,
            "y1": 100,
            "angle": 0,
            "wire_list": [
                {
                    "type": "connection",
                    "connector_obj": "out",
                    "wire_obj": "wire1",
                    "wire_end": "begin"
                }
            ]
        },
        {
            "type": "resistor",
            "x1": 200,
            "y1": 100,

```

```
"angle": 0,
"resistance": 75,
"wire_list": [
    {
        "type": "connection",
        "connector_obj": "in1",
        "wire_obj": "wire1",
        "wire_end": "end"
    },
    {
        "type": "connection",
        "connector_obj": "out",
        "wire_obj": "wire2",
        "wire_end": "begin"
    }
],
{
    "type": "inductor",
    "x1": 300,
    "y1": 100,
    "angle": 0,
    "inductance": 5,
    "wire_list": [
        {
            "type": "connection",
            "connector_obj": "in1",
            "wire_obj": "wire2",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "connector_obj": "out",
            "wire_obj": "wire3",
            "wire_end": "begin"
        }
    ]
},
{
    "type": "capacitor",
    "x1": 400,
    "y1": 100,
    "angle": 0,
    "capacitance": 1,
    "wire_list": [
        {

```

```
        "type": "connection",
        "connector_obj": "in1",
        "wire_obj": "wire3",
        "wire_end": "end"
    },
    {
        "type": "connection",
        "connector_obj": "out",
        "wire_obj": "wire4",
        "wire_end": "begin"
    }
]
},
{
    "type": "outport",
    "x1": 500,
    "y1": 100,
    "angle": 0,
    "wire_list": [
        {
            "type": "connection",
            "connector_obj": "in1",
            "wire_obj": "wire4",
            "wire_end": "end"
        }
    ]
},
{
    "type": "straight",
    "x1": 120,
    "y1": 100,
    "x2": 170,
    "y2": 100
},
{
    "type": "straight",
    "x1": 230,
    "y1": 100,
    "x2": 270,
    "y2": 100
},
{
    "type": "straight",
    "x1": 330,
    "y1": 100,
    "x2": 370,
```

```
        "y2": 100
    },
    {
        "type": "straight",
        "x1": 430,
        "y1": 100,
        "x2": 480,
        "y2": 100
    }
]
},
{
    "conn_list": [
        [
            [
                [
                    "import",
                    0
                ],
                [
                    "resistor",
                    0
                ]
            ],
            [
                [
                    [
                        [
                            "resistor",
                            1
                        ],
                        [
                            "inductor",
                            0
                        ]
                    ],
                    0
                ],
                [
                    [
                        [
                            "inductor",
                            1
                        ],
                        [
                            "capacitor",
                            0
                        ]
                    ],
                    [
                        [
                            "capacitor",
                            0
                        ]
                    ],
                    [
                        [
                            "capacitor",
                            0
                        ]
                    ]
                ]
            ]
        ]
    ]
}
```

```
    1
  ],
  [
    "outport",
    0
  ]
]
}
```

RLC File Save & Load

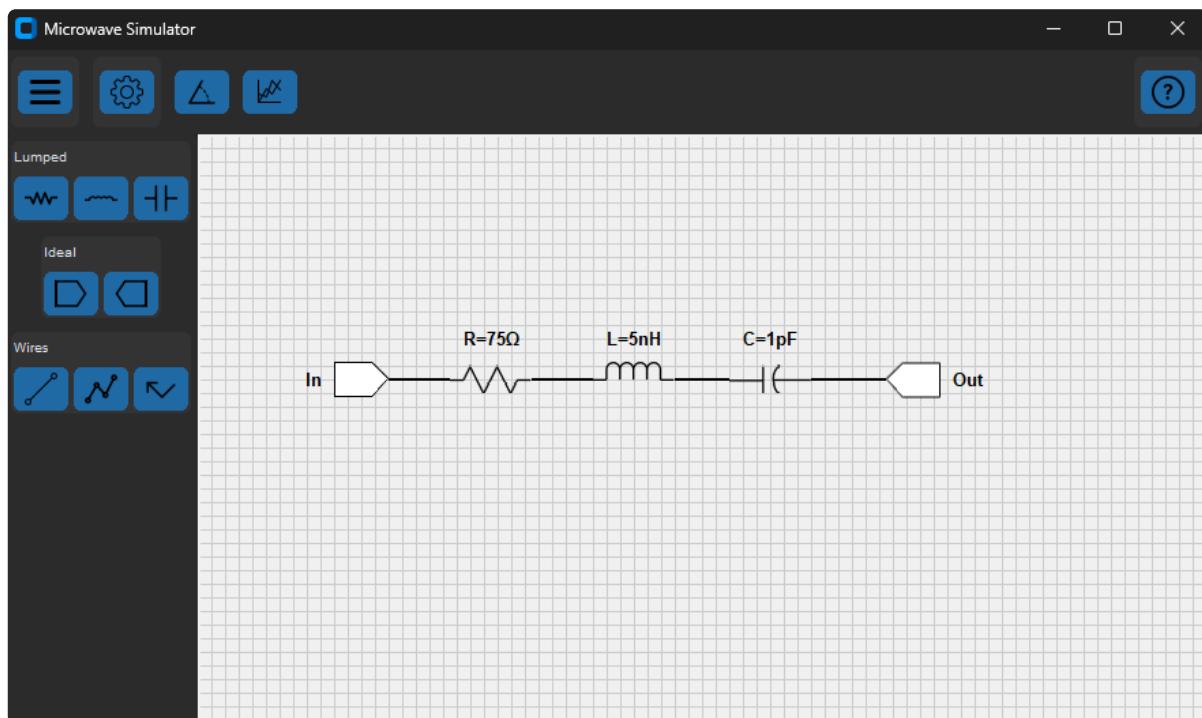


Figure 1

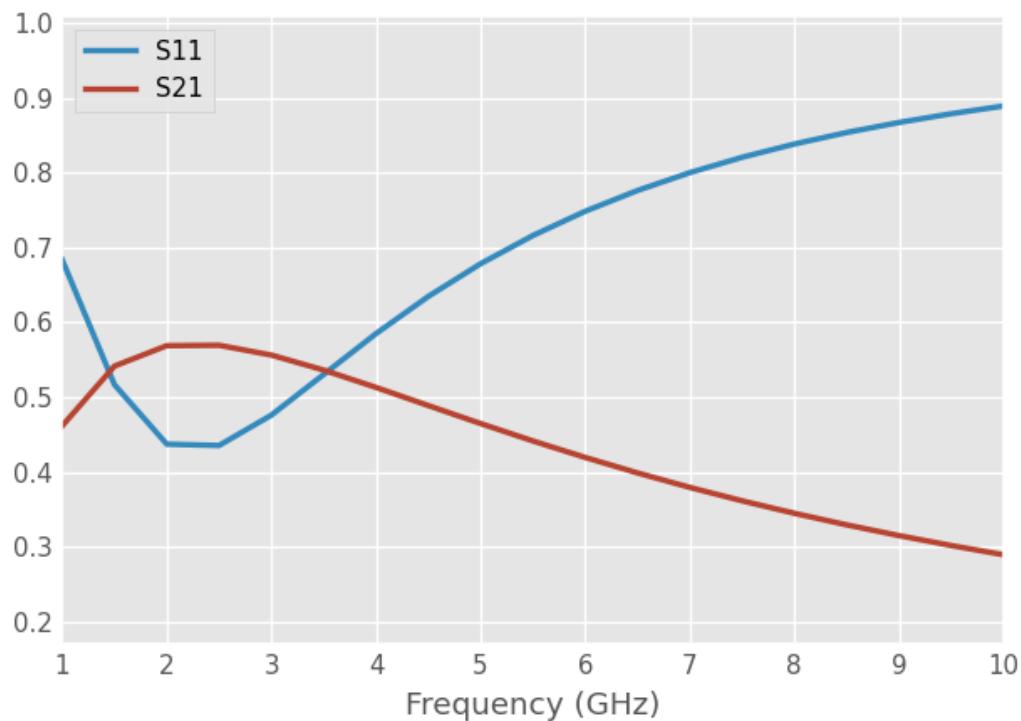
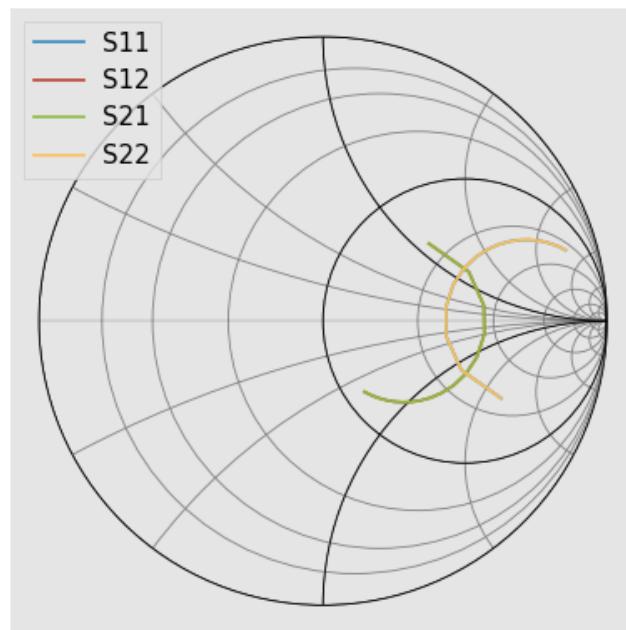


Figure 1



Console output

```
D:\EETools\MicrowaveSimulator\venv\Scripts\python.exe
D:\EETools\MicrowaveSimulator\Sandbox\mini_microwave_simulator.py
[[[0.68368911 0.46154922]
 [0.46154922 0.68368911]]

 [[0.51666132 0.54150201]
 [0.54150201 0.51666132]]

 [[0.43712731 0.56883028]
 [0.56883028 0.43712731]]

 [[0.43535197 0.56937463]
 [0.56937463 0.43535197]]

 [[0.47595789 0.55622445]
 [0.55622445 0.47595789]]

 [[0.53033858 0.5361869 ]
 [0.5361869 0.53033858]]

 [[0.58489041 0.51299248]
 [0.51299248 0.58489041]]

 [[0.63466307 0.48875466]
 [0.48875466 0.63466307]]

 [[0.67836989 0.46467808]
 [0.46467808 0.67836989]]

 [[0.71612899 0.4414337 ]
 [0.4414337 0.71612899]]

 [[0.7485435 0.41937221]
 [0.41937221 0.7485435 ]]

 [[0.77633296 0.39865129]
 [0.39865129 0.77633296]]
```

```
[[0.80018956 0.37931342]
 [0.37931342 0.80018956]]

[[0.82072802 0.36133393]
 [0.36133393 0.82072802]]

[[0.83847445 0.34465089]
 [0.34465089 0.83847445]]

[[0.85387096 0.32918346]
 [0.32918346 0.85387096]]

[[0.86728573 0.31484312]
 [0.31484312 0.86728573]]

[[0.87902421 0.30154034]
 [0.30154034 0.87902421]]

[[0.88933967 0.28918848]
 [0.28918848 0.88933967]]]
```

UI_Lib/wire_button_frame.py

```
import customtkinter as ctk
from pathlib import Path
from PIL import Image

from Wire_Lib import StraightWire, SegmentWire, ElbowWire


class WireButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.wire = None
        self.wire_count = 0 # Added variable to keep track of wire number
count

        # Add frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Wires", font=("Helvetica",
```

```

    10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
        padx=2, pady=2)

        straight_wire_image = ctk.CTkImage(light_image=Image.open
            (Path(__file__).parent /
        "../icons/straight_line.png"),
            dark_image=Image.open
            (Path(__file__).parent /
        "../icons/straight_line.png"),
            size=(24, 24))

        straight_wire_button = ctk.CTkButton(self, text="",
        image=straight_wire_image, width=30,
            command=self.create_straight_wire)
        straight_wire_button.grid(row=1, column=0, sticky=ctk.W, padx=2,
        pady=2)

        segment_wire_image = ctk.CTkImage(light_image=Image.open
            (Path(__file__).parent /
        "../icons/segment_line.png"),
            dark_image=Image.open
            (Path(__file__).parent /
        "../icons/segment_line.png"),
            size=(24, 24))

        segment_wire_button = ctk.CTkButton(self, text="",
        image=segment_wire_image, width=30,
            command=self.create_segment_wire)
        segment_wire_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)

        elbow_wire_image = ctk.CTkImage(light_image=Image.open
            (Path(__file__).parent /
        "../icons/elbow_line.png"),
            dark_image=Image.open
            (Path(__file__).parent /
        "../icons/elbow_line.png"),
            size=(24, 24))

        elbow_wire_button = ctk.CTkButton(self, text="",
        image=elbow_wire_image, width=30,
            command=self.create_elbow_wire)
        elbow_wire_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)

        # Shape button handlers
        def create_straight_wire(self):

```

```

        wire = StraightWire(self.canvas, 0, 0, 0, 0)
        self.create_wire(wire)

    def create_segment_wire(self):
        wire = SegmentWire(self.canvas, 0, 0, 0, 0)
        self.create_wire(wire)

    def create_elbow_wire(self):
        wire = ElbowWire(self.canvas, 0, 0, 0, 0)
        self.create_wire(wire)

    def create_wire(self, wire):
        self.assign_wire_name(wire) # Add method call to assign a wire name
        self.canvas.mouse.current_wire_obj = wire
        self.canvas.show_connectors()
        self.canvas.comp_list.append(wire)
        self.canvas.mouse.draw_wire_mouse_events()

    def assign_wire_name(self, wire): # Added method to create and assign a
wire name
        self.wire_count += 1
        wire_name = 'wire' + str(self.wire_count)
        wire.name = wire_name
        self.canvas.wire_dict[wire_name] = wire

```

UI_Lib/file_menu_frame.py

```

import customtkinter as ctk
from tkinter import filedialog as fd
from pathlib import Path
import json
from PIL import Image

from Comp_Lib import Resistor, Inductor, Capacitor, Connection
from Comp_Lib import Import, Outport
from Wire_Lib import StraightWire, SegmentWire, ElbowWire


class Encoder(json.JSONEncoder):
    def default(self, o):
        if hasattr(o, "reprJson"):
            return o.reprJson()
        else:

```

```
        return super().default(o)

class Decoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self, object_hook=Decoder.from_dict)

    @staticmethod
    def from_dict(_d):
        return _d


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.obj_type_dict = {'resistor': Resistor,
                             'inductor': Inductor,
                             'capacitor': Capacitor,
                             'inport': Inport,
                             'outport': Outport,
                             'straight': StraightWire,
                             'segment': SegmentWire,
                             'elbow': ElbowWire}

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
                               command=self.new_diagram)
        new_btn.pack(pady=5)

        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
                               command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
                               command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
                               command=window.destroy)
        exit_btn.pack(pady=5)
```

```

        my_image = ctk.CTkImage(light_image=Image.open
                                (Path(__file__).parent /
                                 '../icons/hamburger_menu.png'),
                                dark_image=Image.open
                                (Path(__file__).parent /
                                 '../icons/hamburger_menu.png'),
                                size=(24, 24))

    button = ctk.CTkButton(self, text="", image=my_image, width=30,
                           command=self.show_menu)
    button.pack(side=ctk.LEFT, padx=5, pady=10)

def new_diagram(self):
    self.canvas.delete("all")
    self.canvas.comp_list.clear()
    self.canvas.wire_list.clear()

def load_diagram(self):
    try:
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
        with open(f) as file:
            d = json.load(file)
            self.convert_json_data(d)
    except FileNotFoundError:
        with open('untitled.canvas', 'w') as _file:
            pass

def convert_json_data(self, data):
    """Convert json data to a circuit object"""
    # Get circuit list from json data
    json_comp_list = data[0]['comp_list']
    for json_comp in json_comp_list:
        if json_comp['type'] == 'resistor':
            res = Resistor(self.canvas, int(json_comp['x1']),
                           int(json_comp['y1']), int(json_comp['resistance']))
            conn_dict = json_comp['wire_list'][0]
            res.wire_list.append(Connection(conn_dict['comp_conn'],
                                             conn_dict['wire_name'],
                                             conn_dict['wire_end']))
            conn_dict = json_comp['wire_list'][1]
            res.wire_list.append(Connection(conn_dict['comp_conn'],
                                             conn_dict['wire_name'],
                                             conn_dict['wire_end']))
            self.canvas.comp_list.append(res)
        elif json_comp['type'] == 'inductor':

```

```

        ind = Inductor(self.canvas, int(json_comp['x1']),
int(json_comp['y1']), int(json_comp['inductance']))
        conn_dict = json_comp['wire_list'][0]
        ind.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
                                         conn_dict['wire_end']))
        conn_dict = json_comp['wire_list'][1]
        ind.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
                                         conn_dict['wire_end']))
        self.canvas.comp_list.append(ind)
    elif json_comp['type'] == 'capacitor':
        cap = Capacitor(self.canvas, int(json_comp['x1']),
int(json_comp['y1']), int(json_comp['capacitance']))
        conn_dict = json_comp['wire_list'][0]
        cap.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
                                         conn_dict['wire_end']))
        conn_dict = json_comp['wire_list'][1]
        cap.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
                                         conn_dict['wire_end']))
        self.canvas.comp_list.append(cap)
    elif json_comp['type'] == 'inport':
        port1 = Inport(self.canvas, int(json_comp['x1']),
int(json_comp['y1']))
        conn_dict = json_comp['wire_list'][0]
        port1.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
                                         conn_dict['wire_end']))
        self.canvas.comp_list.append(port1)
    elif json_comp['type'] == 'outport':
        port2 = Outport(self.canvas, int(json_comp['x1']),
int(json_comp['y1']))
        conn_dict = json_comp['wire_list'][0]
        port2.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
                                         conn_dict['wire_end']))
        self.canvas.comp_list.append(port2)
    elif json_comp['type'] == 'straight':
        wire = StraightWire(self.canvas, int(json_comp['x1']),
int(json_comp['y1']), int(json_comp['x2']),
                                         int(json_comp['y2']))
        wire.name = json_comp['name']
        self.canvas.comp_list.append(wire)
        self.canvas.wire_dict[wire.name] = wire

```

```

# Get connection list from json data
self.canvas.conn_list = data[1]['conn_list']

def save_diagram(self):
    comp_dict = {'comp_list': self.canvas.comp_list}
    conn_dict = {'conn_list': self.canvas.conn_list}
    circuit = [comp_dict, conn_dict]

    filetypes = (('json files', '*.json'), ('All files', '*.*'))
    f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
    with open(f, 'w') as file:
        file.write(json.dumps(circuit, cls=Encoder, indent=4))

def show_menu(self):
    if not self.menu_on:
        self.menu_frame.place(x=5, y=60)
        self.menu_frame.tkraise()
        self.menu_on = True
    else:
        self.menu_frame.place_forget()
        self.menu_on = False

```

Circuits/RLC.json

```
[
{
    "comp_list": [
        {
            "type": "inport",
            "x1": 120,
            "y1": 180,
            "angle": 0,
            "wire_list": [
                {
                    "type": "connection",
                    "comp_conn": "out",
                    "wire_name": "wire1",
                    "wire_end": "begin"
                }
            ]
        },
        {

```

```
        "type": "resistor",
        "x1": 215,
        "y1": 180,
        "angle": 0,
        "resistance": 75,
        "wire_list": [
            {
                "type": "connection",
                "comp_conn": "in1",
                "wire_name": "wire1",
                "wire_end": "end"
            },
            {
                "type": "connection",
                "comp_conn": "out",
                "wire_name": "wire2",
                "wire_end": "begin"
            }
        ]
    },
    {
        "type": "inductor",
        "x1": 320,
        "y1": 180,
        "angle": 0,
        "inductance": 5,
        "wire_list": [
            {
                "type": "connection",
                "comp_conn": "in1",
                "wire_name": "wire2",
                "wire_end": "end"
            },
            {
                "type": "connection",
                "comp_conn": "out",
                "wire_name": "wire3",
                "wire_end": "begin"
            }
        ]
    },
    {
        "type": "capacitor",
        "x1": 420,
        "y1": 180,
        "angle": 0,
```

```
"capacitance": 1,  
"wire_list": [  
    {  
        "type": "connection",  
        "comp_conn": "in1",  
        "wire_name": "wire3",  
        "wire_end": "end"  
    },  
    {  
        "type": "connection",  
        "comp_conn": "out",  
        "wire_name": "wire4",  
        "wire_end": "begin"  
    },  
    {  
        "type": "connection",  
        "comp_conn": "out",  
        "wire_name": "wire4",  
        "wire_end": "end"  
    },  
    {  
        "type": "connection",  
        "comp_conn": "out",  
        "wire_name": "wire5",  
        "wire_end": "begin"  
    }  
],  
,  
{  
    "type": "outport",  
    "x1": 525,  
    "y1": 180,  
    "angle": 0,  
    "wire_list": [  
        {  
            "type": "connection",  
            "comp_conn": "in1",  
            "wire_name": "wire5",  
            "wire_end": "end"  
        }  
    ]  
,  
{  
    "type": "straight",  
    "x1": 140.0,  
    "y1": 180.0,
```

```
        "x2": 185.0,
        "y2": 180.0,
        "name": "wire1"
    },
    {
        "type": "straight",
        "x1": 245.0,
        "y1": 180.0,
        "x2": 290.0,
        "y2": 180.0,
        "name": "wire2"
    },
    {
        "type": "straight",
        "x1": 350.0,
        "y1": 180.0,
        "x2": 390.0,
        "y2": 180.0,
        "name": "wire3"
    },
    {
        "type": "straight",
        "x1": 450.0,
        "y1": 180.0,
        "x2": 450.0,
        "y2": 180.0,
        "name": "wire4"
    },
    {
        "type": "straight",
        "x1": 450.0,
        "y1": 180.0,
        "x2": 505.0,
        "y2": 180.0,
        "name": "wire5"
    }
]
},
{
    "conn_list": [
        [
            [
                "inport",
                0
            ],
            [

```

```

        "resistor",
        0
    ],
    [
        [
            [
                "resistor",
                1
            ],
            [
                [
                    "inductor",
                    0
                ]
            ],
            [
                [
                    [
                        [
                            "inductor",
                            1
                        ],
                        [
                            [
                                "capacitor",
                                0
                            ]
                        ]
                    ],
                    [
                        [
                            [
                                [
                                    [
                                        [
                                            "capacitor",
                                            1
                                        ],
                                        [
                                            [
                                                "outport",
                                                0
                                            ]
                                        ]
                                    ]
                                ]
                            ]
                        ]
                    ]
                }
            ]
        ]
    ]
]

```

Summary

Microwave circuit simulation using the Scikit-RF library provides a good basis for RF and microwave simulators. This concludes the development of our third advanced electrical engineering project. The final project will be an Analog Circuit Simulator using many of the

techniques we development for the Microwave Circuit Simulator. If you made it this far, you are ready to complete the last tool in the EE Tools software. See you in the next chapter.

Chapter 8 - Analog Circuit Simulator

Design, Features, and Specifications

Analog Simulator allows the design of basic electronic circuits and analysis of voltage, current, and power in the circuit. Spice is the standard library for analog circuit analysis and fortunately there is a Python library called PySpice.

Program Design & Architecture:

- ✓ Modify the Microwave Simulator to create the Analog Simulator
- ✓ Object-Oriented Programming (OOP) - class abstraction, inheritance, & polymorphism
- ✓ KISS - Keep it simple, silly
- ✓ DRY - Don't repeat yourself
- ✓ SOC - Separation of concerns
- ✓ Python Modules & Packages
- ✓ User-interface
 - ✓ TopFrame class
 - ✓ File menu frame
 - ✓ Settings menu frame
 - ✓ Rotation button
 - ✓ Help menu frame
 - ✓ Analysis button
 - ✓ Left Frame Class with Circuit Component Menus
 - ✓ Canvas class
 - ✓ Mouse class
- ✓ Circuit components
 - ✓ Components
 - ✓ Resistor
 - ✓ Capacitor
 - ✓ Inductor
 - ✓ Transistor
 - ✓ Sources
 - ✓ Voltage source
 - ✓ Current source

- ✓ Wires
 - ✓ Straight Wire Class
 - ✓ Segment Wire Class
 - ✓ Elbow Wire Class
- ✓ Grid class
- ✓ Analysis
 - ✓ Raw data display
 - ✓ Line graphs

Key Technologies Needed:

- ✓ Analog analysis library - PySpice

Project Setup

Language: Python 3.11

IDE: PyCharm 2023.2.1 (Community Edition)

Project directory: D:/EETools/AnalogSimulator

Graphics library: CustomTkinter (<https://customtkinter.tomschimansky.com/>)

External libraries:

- ✓ pip install customtkinter
- ✓ python.exe -m pip install --upgrade pip
- ✓ pip install ctkcolorpicker
- ✓ pip install tkinter-tooltip
- ✓ pip install pyInstaller - Create .exe file
- ✓ pip install matplotlib
- ✓ Add images and icons directories to the project.

Analog Analysis Library

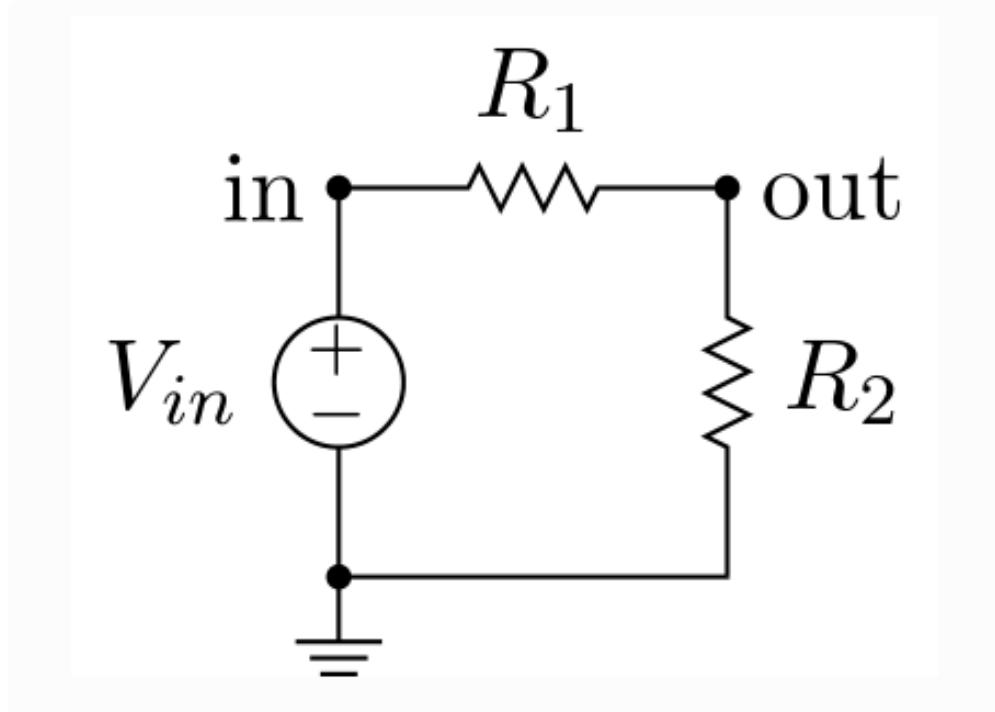
- [PySpice](#) and [Ngspice](#)
- ✓ pip install pyspice
- ✓ `pyspice-post-installation --install-ngspice-dll`

PySpice Evaluation

[Home Page](#)

[Documentation](#)

[GitHub Repository](#)



Sandbox/pyspice_voltage_divider.py

```
import PySpice.Logging.Logging as Logging
logger = Logging.setup_logging()
from PySpice.Spice.Netlist import Circuit

voltage_unit = u_V = 1
resistance_unit = u_kΩ = 1000

circuit = Circuit('Voltage Divider')

circuit.V('input', 'in', circuit.gnd, 10*u_V)
circuit.R(1, 'in', 'out', 9*u_kΩ)
circuit.R(2, 'out', circuit.gnd, 1*u_kΩ)

simulator = circuit.simulator(temperature=25, nominal_temperature=25)

analysis = simulator.operating_point()
for node in (analysis['in'], analysis.out): # .in is invalid !
```

```
print('Node {}: {} V'.format(str(node), float(node)))

print()

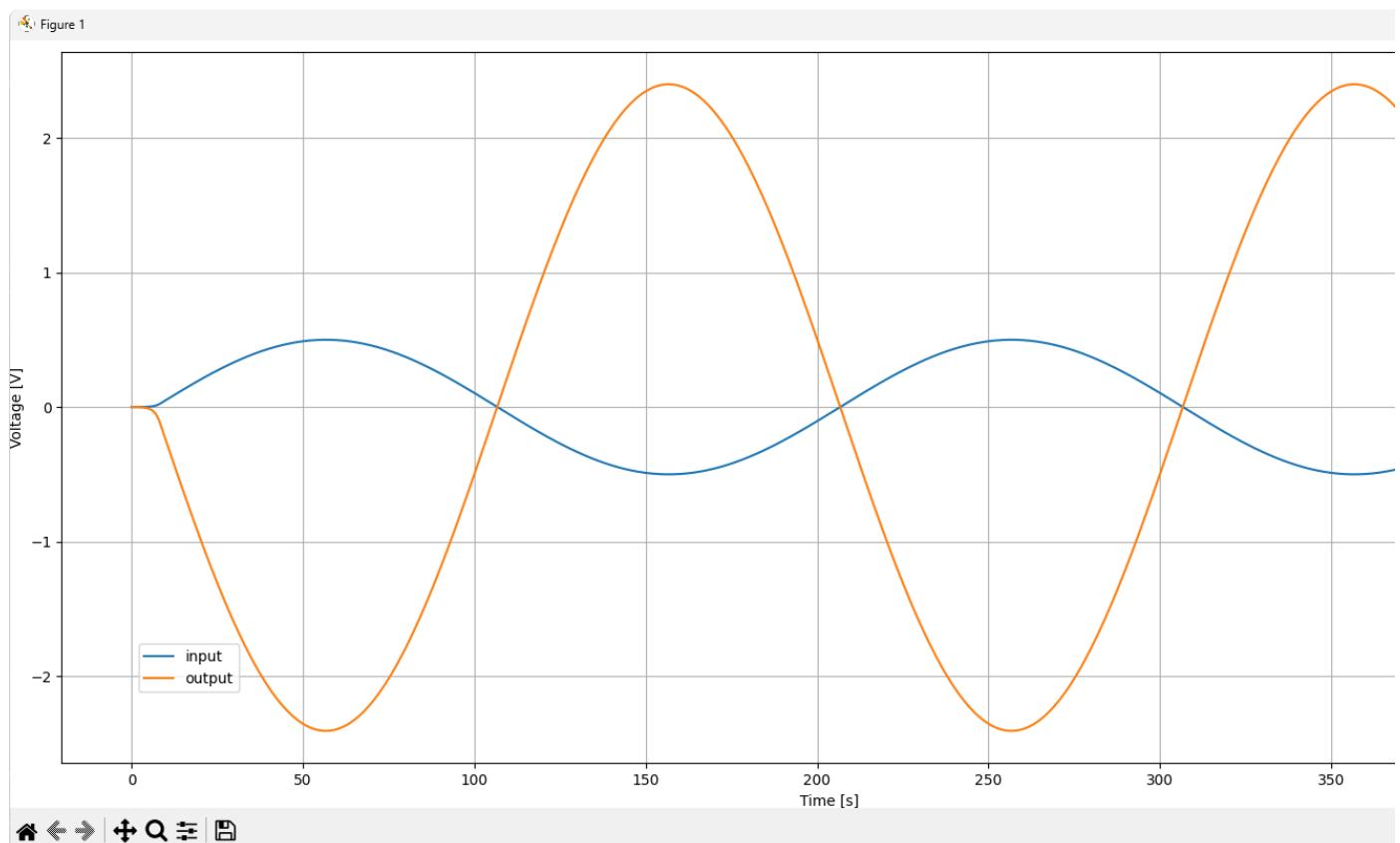
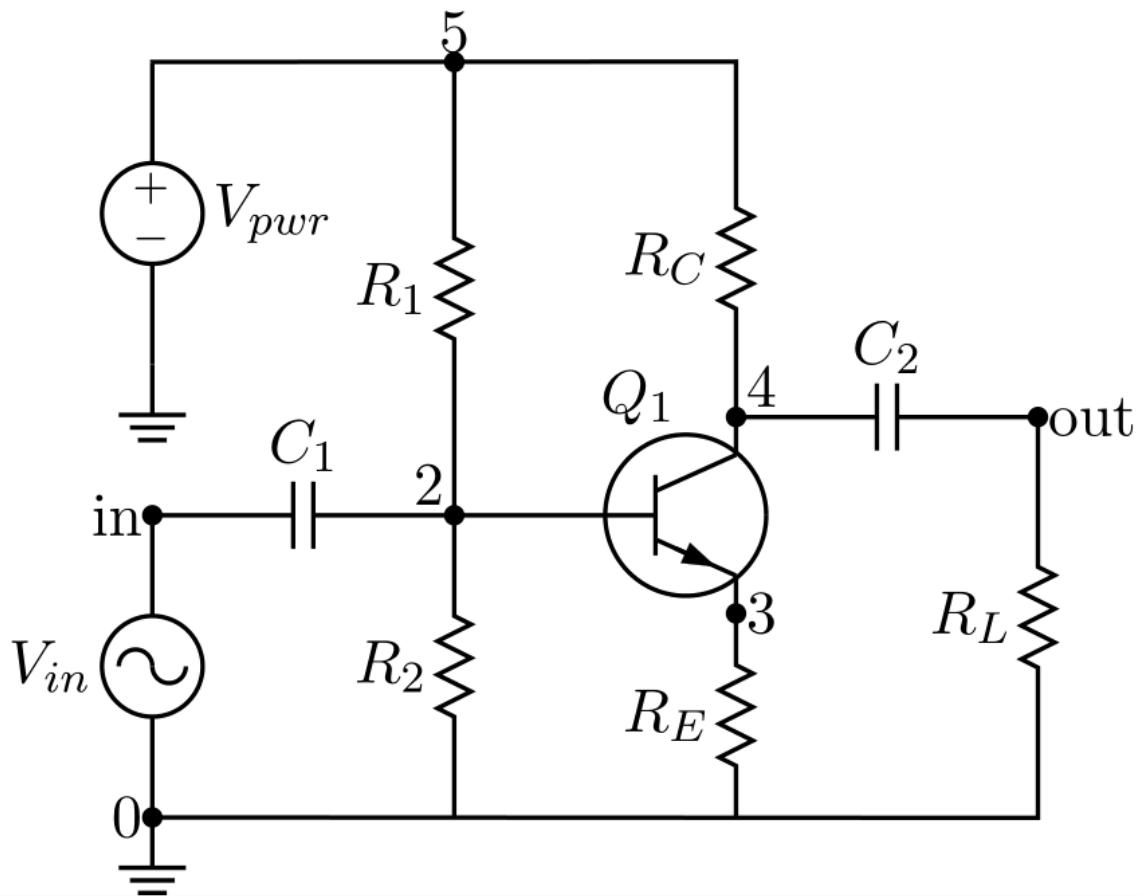
analysis = simulator.dc_sensitivity('v(out)')
for element in analysis.elements.values():
    print(element, float(element))
```

Console output

```
Node in: 10.0 V
Node out: 1.0 V

vinput 0.09999999999621426
r1_scale -0.8999991000992625
r1_bv_max -0.0
r1_m 0.8999999999287899
r1_w -0.0
r1_l -0.0
r1 -9.999990001102918e-05
r1:ef -0.0
r1:wf -0.0
r1:lf -0.0
r1:bv_max -0.0
r2_scale 0.899999099855317
r2_bv_max -0.0
r2_m -0.899999999165925
r2_w -0.0
r2_l -0.0
r2 0.0008999991000504734
r2:ef -0.0
r2:wf -0.0
r2:lf -0.0
r2:bv_max -0.0
```

AC Coupled Transistor Amplifier



```

import matplotlib.pyplot as plt
import PySpice.Logging.Logging as Logging
logger = Logging.setup_logging()
from PySpice.Doc.ExampleTools import find_libraries
from PySpice.Spice.Library import SpiceLibrary
from PySpice.Spice.Netlist import Circuit


libraries_path = find_libraries()
spice_library = SpiceLibrary(libraries_path)

voltage_unit = u_V = 1
resistance_unit = u_kΩ = 1000
frequency_unit = u_kHz = 1000
capacitance_unit = u_uF = 1e-6
u_MΩ = 1e6
u_pF = 1e-12

circuit = Circuit('Transistor')

circuit.V('power', 5, circuit.gnd, 15*u_V)
source = circuit.SinusoidalVoltageSource('in', 'in', circuit.gnd,
amplitude=.5*u_V, frequency=1*u_kHz)
circuit.C(1, 'in', 2, 10*u_uF)
circuit.R(1, 5, 2, 100*u_kΩ)
circuit.R(2, 2, 0, 20*u_kΩ)
circuit.R('C', 5, 4, 10*u_kΩ)
circuit.BJT(1, 4, 2, 3, model='bjt') # Q is mapped to BJT !
circuit.model('bjt', 'npn', bf=80, cjc=5*u_pF, rb=100)
circuit.R('E', 3, 0, 2*u_kΩ)
circuit.C(2, 4, 'out', 10*u_uF)
circuit.R('Load', 'out', 0, 1*u_MΩ)

figure, ax = plt.subplots(figsize=(20, 10))

# .ac dec 5 10m 1G

simulator = circuit.simulator(temperature=25, nominal_temperature=25)
analysis = simulator.transient(step_time=source.period/200,
end_time=source.period*2)

ax.set_title('')
ax.set_xlabel('Time [s]')
ax.set_ylabel('Voltage [V]')
ax.grid()

```

```
ax.plot(analysis['in'])
ax.plot(analysis.out)
ax.legend(('input', 'output'), loc=(.05,.1))

plt.tight_layout()
plt.show()
```

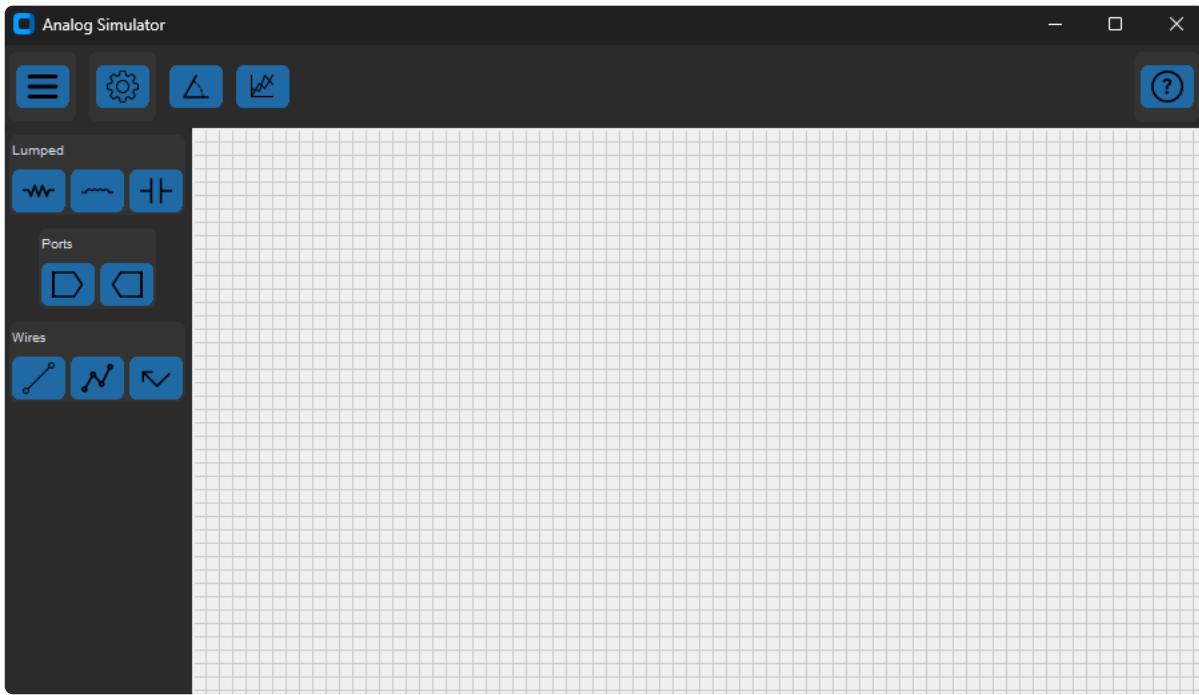
Analog Simulator User Interface

Directories copied from Microwave Simulator

- Circuits
 - Deleted circuit .json files
- Comp_Lib
- Helper_Lib
- icons
- images
- UI_Lib
- Wire_Lib

Modifications:

- Changed name of ideal_button_frame.py to ports_button_frame.py
- Changed name of `IdealButtonFrame` class to `PortsButtonFrame`
- Modified `left_frame.py` to create a `PortsButtonFrame` instead of `IdealButtonFrame`
- Modifications to `top_frame.py`
 - Removed `skrf` import
 - Replaced all code in `analyze_circuit()` method with pass - this is where we will put PySpice analysis code



Looks a lot like the Microwave Simulator user interface. Changes needed:

- Add voltage and current sources
- Add nodes for testing voltage
- Add a Ground Class
- Add PySpice analysis to Top Frame class

Refactor Classes

Universal Connector Method

Define common connector locations for 1-port and 2-port components

1-Port Components

- Ground

2-Port Components

- Resistor
- Capacitor
- Inductor
- Voltage Source

- Current Source

Universal Component Class

Base class: Component

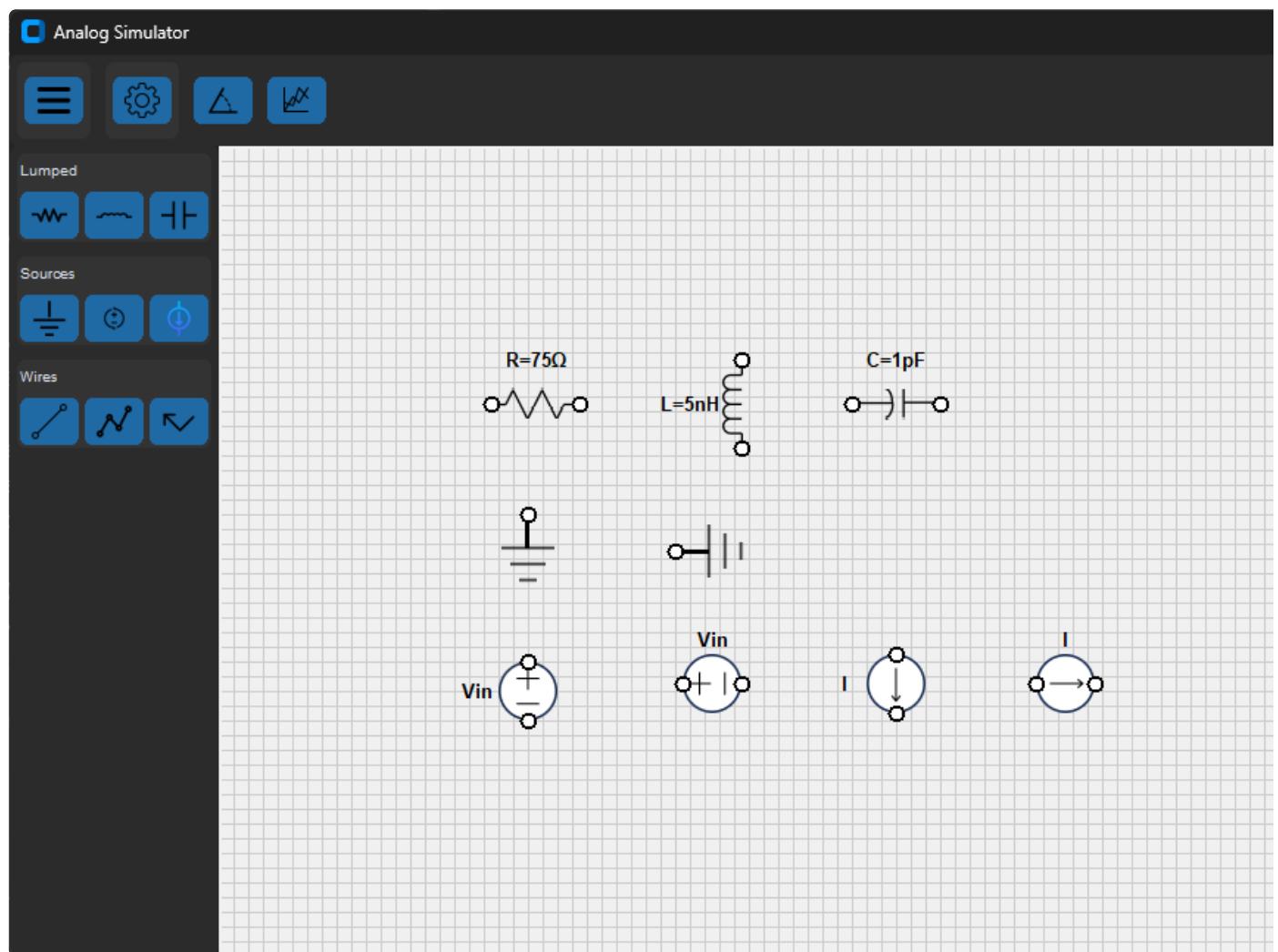
Universal derived class: AnalogComponent

Universal Wire Class

Base class: Wire

Universal derived class: AnalogWire

Universal Component Class with Universal Connector Method



Comp_Lib/component.py

```
import tkinter as tk
from PIL import Image, ImageTk
from pathlib import Path

from Helper_Lib import Point


class Component:
    def __init__(self, canvas, comp_type, x1, y1, value):
        """Base class for component classes"""
        self.canvas = canvas
        self.comp_type = comp_type
        self.x1 = x1
        self.y1 = y1
        self.value = value

        self.id = None
        self.sel_id = None

        self.is_selected = False
        self.is_drawing = False
        self.selector = None
        self.angle = 0
        self.comp_text = None
        self.text = None
        self.text_id = None

        self.filename = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None

        self.out = None
        self.in1 = None

        self.conn_list = []
        self.wire_list = []

    # Define component parameters in a dictionary of dictionaries
    self.params = {
        'filename': {
            'resistor': '../images/lumped/resistor_60x30.png',
            'capacitor': '../images/lumped/capacitor_60x30.png',
            'inductor': '../images/lumped/inductor_60x30.png',
            'ground': '../images/sources/ground_50x40.png',
            'vsource': '../images/sources/voltage_source_40x40.png',
        }
    }
```

```

        'isource': '../images/sources/current_source_40x40.png"
    },
    'text': {
        'resistor': 'R=' + str(self.value) + '\u0336',
        'capacitor': 'C=' + str(self.value) + 'pF',
        'inductor': 'L=' + str(self.value) + 'nH',
        'ground': "",
        'vsource': "Vin",
        'isource': "I"
    }
}

def set_image_filename(self):
    self.filename = Path(__file__).parent / self.params['filename']
    [self.comp_type]

def create_text(self):
    self.comp_text = self.params['text'][self.comp_type]
    if self.comp_type == 'isource' or self.comp_type == 'vsource':
        text_loc = Point(self.x1-35, self.y1) # Put text on left side of
symbol
    else:
        text_loc = Point(self.x1, self.y1-30) # Put text above symbol
    self.text_id = self.canvas.create_text(text_loc.x, text_loc.y,
                                           text=self.comp_text, fill="black",
                                           font='Helvetica 10 bold',
                                           angle=self.angle, tags="text")

def create_image(self, filename):
    """Initial component image creation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True)
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

def update_position(self):
    """Update the position when the gate object is moved"""
    self.canvas.coords(self.id, self.x1, self.y1) # Update position

def update_image(self, filename):
    """Update the image for gate symbol rotation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True) # Update
image rotation
    self.ph_image = ImageTk.PhotoImage(self.a_image)

```

```

        self.canvas.itemconfig(self.id, image=self.ph_image) # Update image

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

    def update_text(self):
        if self.comp_type == 'isource' or self.comp_type == 'vsource':
            if self.angle == 0 or self.angle == 180:
                self.set_east_text()
            elif self.angle == 90 or self.angle == 270:
                self.set_north_text()
        else:
            if self.angle == 0 or self.angle == 180:
                self.set_north_text()
            elif self.angle == 90 or self.angle == 270:
                self.set_east_text()

    def set_east_text(self):
        self.canvas.coords(self.text_id, self.x1 - 35, self.y1)

    def set_north_text(self):
        self.canvas.coords(self.text_id, self.x1, self.y1 - 30)

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
        self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
        outline="red", width=2)
        self.set_selector_visibility()

    def update_selector(self):
        """Update the red rectangle selector coordinates and check to see if the gate is selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
        self.bbox[3] + 5
        self.canvas.coords(self.sel_id, x1, y1, x2, y2)
        self.set_selector_visibility()

    def set_selector_visibility(self):
        """Set the selector visibility state"""
        if self.is_selected:
            self.canvas.itemconfig(self.sel_id, state='normal')
        else:

```

```

        self.canvas.itemconfig(self.sel_id, state='hidden')

def set_connector_visibility(self):
    """Set the connector visibility state"""
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

def check_connector_hit(self, x, y):
    """Hit test to see if a connector is at the provided x, y
coordinates"""
    for conn in self.conn_list:
        if conn.connector_hit_test(x, y):
            return conn
    return None

def move_connected_wires(self):
    """Resize connected wires if the shape is moved"""
    for connection in self.wire_list: # comp_conn, wire_name, wire_end
        for connector in self.conn_list:
            if connector.name == connection.comp_conn:
                wire_obj = self.canvas.wire_dict[connection.wire_name]
                if connection.wire_end == "begin":
                    wire_obj.x1 = connector.x
                    wire_obj.y1 = connector.y
                elif connection.wire_end == "end":
                    wire_obj.x2 = connector.x
                    wire_obj.y2 = connector.y

def rotate(self):
    """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
if angle > 270 deg"""
    self.angle += 90
    if self.angle > 270:
        self.angle = 0

def hit_test(self, x, y):
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    if x1 <= x <= x2 and y1 <= y <= y2:
        self.is_selected = True
    else:
        self.is_selected = False

```

Comp_Lib/analog_component.py

```
from Comp_Lib.component import Component
from Comp_Lib.connector import Connector
from Helper_Lib import Point


class AnalogComponent(Component):
    """Universal model for analog components"""
    def __init__(self, canvas, comp_type, x1, y1, value=0):
        super().__init__(canvas, comp_type, x1, y1, value)

        self.conn_params = {
            'ports': {
                'resistor': 2,
                'capacitor': 2,
                'inductor': 2,
                'ground': 1,
                'vsource': 2,
                'isource': 2
            },
            'conn_loc': {
                'resistor': 'ew',
                'capacitor': 'ew',
                'inductor': 'ew',
                'ground': 'n',
                'vsource': 'ns',
                'isource': 'ns'
            }
        }

        self.create()

    def create(self):
        self.set_image_filename()
        self.create_image(self.filename)
        self.update_bbox()
        self.create_text()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
```

```

    self.update_bbox()
    self.update_text()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def create_connectors(self):
    # Calculate position of connectors from current comp position and size
    center, e, w, n, s = self.get_geometry()

    num_ports = self.conn_params['ports'][self.comp_type]
    if num_ports == 1:
        self.out = Connector(self.canvas, "in1", center.x, center.y)
        self.conn_list = [self.out]
    elif num_ports == 2:
        self.out = Connector(self.canvas, "out", center.x, center.y)
        self.in1 = Connector(self.canvas, "in1", center.x, center.y)
        self.conn_list = [self.in1, self.out]

def update_connectors(self):
    """Update the position of connectors here"""
    center, e, w, n, s = self.get_geometry()

    conn_loc = self.conn_params['conn_loc'][self.comp_type]
    if conn_loc == 'ew': # 2-port with ew ports
        self.calc_ew_conn_rotation(n, s, e, w)
    elif conn_loc == 'ns': # 2-port with ns ports
        self.calc_ns_conn_rotation(n, s, e, w)
    elif conn_loc == 'n': # 1-port with n port
        self.calc_n_conn_rotation(n, w)

    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def calc_ew_conn_rotation(self, n, s, e, w):
    if self.angle == 0 or self.angle == 180:
        self.out.x, self.out.y = w.x, w.y
        self.in1.x, self.in1.y = e.x, e.y
    elif self.angle == 90 or self.angle == 270:
        self.out.x, self.out.y = n.x, n.y
        self.in1.x, self.in1.y = s.x, s.y

def calc_ns_conn_rotation(self, n, s, e, w):
    if self.angle == 0 or self.angle == 180:

```

```

        self.out.x, self.out.y = n.x, n.y
        self.in1.x, self.in1.y = s.x, s.y
    elif self.angle == 90 or self.angle == 270:
        self.out.x, self.out.y = w.x, w.y
        self.in1.x, self.in1.y = e.x, e.y

def calc_n_conn_rotation(self, n, w):
    if self.angle == 0 or self.angle == 180:
        self.out.x, self.out.y = n.x, n.y
    elif self.angle == 90 or self.angle == 270:
        self.out.x, self.out.y = w.x, w.y

def get_geometry(self):
    sign = lambda angle: 1 if angle == 0 or angle == 180 else -1

    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    e = Point(center.x - sign(self.angle) * w / 2, center.y)
    w = Point(center.x + sign(self.angle) * w / 2, center.y)
    n = Point(center.x, center.y - sign(self.angle) * h / 2)
    s = Point(center.x, center.y + sign(self.angle) * h / 2)

    return center, e, w, n, s

def __repr__(self):
    return ("Type: " + self.comp_type + " x1: " + str(self.x1) + " y1: " +
str(self.y1) + " value: " +
            str(self.value) + " wire list: " +
str(self.wire_list.__repr__()))

def reprJson(self):
    return dict(type=self.comp_type, x1=self.x1, y1=self.y1,
angle=self.angle, value=self.value,
            wire_list=self.wire_list)

```

UI_Lib/lump_button_frame.py

```

import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path

```

```

from PIL import Image

from Comp_Lib import AnalogComponent


class LumpButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None

        self.button_list = [("resistor", "../icons/resistor.png"),
                            ("inductor", "../icons/inductor.png"),
                            ("capacitor", "../icons/capacitor.png")]

    def init_frame_widgets(self):
        frame_name_label = ctk.CTkLabel(self, text="Lumped", font=("Helvetica", 10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W, padx=2, pady=2)

        row_num, col_num = 1, 0
        for button in self.button_list:
            a_image = ctk.CTkImage(light_image=Image.open
                                   (Path(__file__).parent / button[1]),
                                   dark_image=Image.open
                                   (Path(__file__).parent / button[1]),
                                   size=(24, 24))
            self.button_id = ctk.CTkButton(self, text="", image=a_image,
                                           width=30,
                                           command=lambda
                                               a_name=button[0]:self.create_events(a_name))
            self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W, padx=2, pady=2)
            ToolTip(self.button_id, msg=button[0])
            row_num, col_num = self.update_grid_numbers(row_num, col_num)

    def create_events(self, name):
        comp = None
        if name == "resistor":
            comp = AnalogComponent(self.canvas, 'resistor', 100, 100, 75)
        elif name == "inductor":
            comp = AnalogComponent(self.canvas, 'inductor', 100, 100, 5)

```

```

        elif name == "capacitor":
            comp = AnalogComponent(self.canvas, 'capacitor', 100, 100, 1)
            self.canvas.comp_list.append(comp)
            self.canvas.redraw()
            self.canvas.mouse.move_mouse_bind_events()

    @staticmethod
    def update_grid_numbers(row, column):
        column += 1
        if column > 2:
            column = 0
        row += 1
        return row, column

```

UI_Lib/sources_button_frame.py

```

import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from Comp_Lib import AnalogComponent


class SourcesButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None

        self.button_list = [("ground", "../icons/ground.png"),
                           ("vsource", "../icons/voltage_source.png"),
                           ("isource", "../icons/current_source.png")]

        self.init_frame_widgets()

    def init_frame_widgets(self):
        frame_name_label = ctk.CTkLabel(self, text="Sources", font=("Helvetica", 10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
                             padx=2, pady=2)

```

```

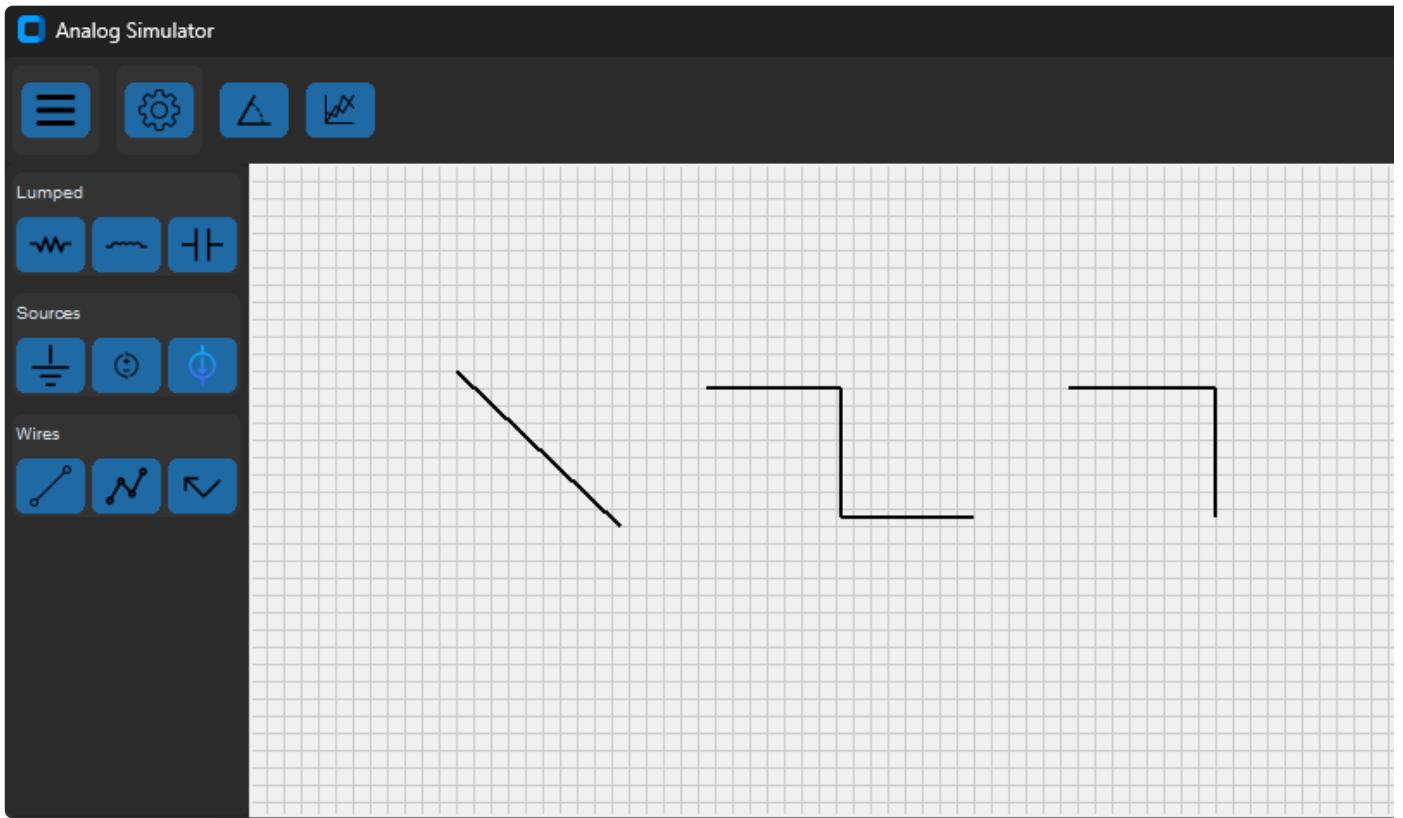
row_num, col_num = 1, 0
for button in self.button_list:
    a_image = ctk.CTkImage(light_image=Image.open
                           (Path(__file__).parent / button[1]),
                           dark_image=Image.open
                           (Path(__file__).parent / button[1]),
                           size=(24, 24))
    self.button_id = ctk.CTkButton(self, text="", image=a_image,
width=30,
                           command=lambda
a_name=button[0]:self.create_events(a_name))
    self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W,
padx=2, pady=2)
    ToolTip(self.button_id, msg=button[0])
    row_num, col_num = self.update_grid_numbers(row_num, col_num)

def create_events(self, name):
    comp = None
    if name == "ground":
        comp = AnalogComponent(self.canvas, 'ground', 100, 100)
    elif name == "vsource":
        comp = AnalogComponent(self.canvas, 'vsource', 100, 100)
    elif name == "isource":
        comp = AnalogComponent(self.canvas, 'isource', 100, 100)
    self.canvas.comp_list.append(comp)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

@staticmethod
def update_grid_numbers(row, column):
    column += 1
    if column > 2:
        column = 0
        row += 1
    return row, column

```

Universal Wire Class



Wire_lib/wire.py

```
from Wire_Lib.wire_selector import WireSelector

class Wire:
    """Base class for wire classes"""
    def __init__(self, canvas, wire_type, x1, y1, x2, y2):
        self.canvas = canvas
        self.wire_type = wire_type
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        self.fill_color = "black"
        self.border_width = 2
        self.line_direction = "horizontal"

        self.name = None
        self.id = None
        self.is_selected = False
        self.selector = None
        self.width = 2
```

```

    self.bbox = None

    # Connections for wire list
    self.in_cnx = None
    self.out_cnx = None
    self.cnx = []

    self.sel1, self.sel2 = None, None

def update_bbox(self):
    self.bbox = self.canvas.bbox(self.id)

def create_selectors(self):
    self.sel1 = WireSelector(self.canvas, "begin", self.x1, self.y1)
    self.sel2 = WireSelector(self.canvas, "end", self.x2, self.y2)

def update_selectors(self):
    self.sel1.x, self.sel1.y = self.x1, self.y1
    self.sel2.x, self.sel2.y = self.x2, self.y2
    self.sel1.update()
    self.sel2.update()

def update_selection(self):
    if self.is_selected:
        self.canvas.itemconfigure(self.id, fill="red")
        self.canvas.itemconfigure(self.sel1.id, state='normal')
        self.canvas.itemconfigure(self.sel2.id, state='normal')
    else:
        self.canvas.itemconfigure(self.id, fill="black")
        self.canvas.itemconfigure(self.sel1.id, state='hidden')
        self.canvas.itemconfigure(self.sel2.id, state='hidden')

def hit_test(self, x, y):
    x1, y1 = self.bbox[0], self.bbox[1]
    x2, y2 = self.bbox[2], self.bbox[3]
    if x1 <= x <= x2 and y1 <= y <= y2:
        self.is_selected = True
    else:
        self.is_selected = False

def sel_hit_test(self, x, y):
    if self.sel1.selector_hit_test(x, y):
        self.selector = self.sel1.name
        return self.sel1
    elif self.sel2.selector_hit_test(x, y):
        self.selector = self.sel2.name

```

```

        return self.sel2
    else:
        return None

def resize(self, offsets, event):
    offset_x1, offset_y1, offset_x2, offset_y2 = offsets
    if self.selector == "end":
        x2 = event.x - offset_x2
        y2 = event.y - offset_y2
        self.x2, self.y2 = x2, y2
        self.x2, self.y2 = self.canvas.grid.snap_to_grid(self.x2, self.y2)
    elif self.selector == "begin":
        x1 = event.x - offset_x1
        y1 = event.y - offset_y1
        self.x1, self.y1 = x1, y1
        self.x1, self.y1 = self.canvas.grid.snap_to_grid(self.x1, self.y1)

def create_wire_list_cnx(self, comp_type, wire_end):
    if wire_end == 'out':
        if comp_type == "inport":
            self.out_cnx = (comp_type, 0)
        else:
            self.out_cnx = (comp_type, 1)
    elif wire_end == 'in1':
        self.in_cnx = (comp_type, 0)
    if self.in_cnx and self.out_cnx:
        self.cnx = [self.out_cnx, self.in_cnx]
        self.canvas.conn_list.append(self.cnx)

```

Wire_Lib/analog_wire.py

```

from Wire_Lib.wire import Wire

class AnalogWire(Wire):
    def __init__(self, canvas, wire_type, x1, y1, x2, y2):
        super().__init__(canvas, wire_type, x1, y1, x2, y2)

        self.seg1, self.seg2, self.seg3 = None, None, None
        self.segment_list = []

        self.create_wire()
        self.update_bbox()

```

```

        self.create_selectors()
        self.update_selection()

    def create_wire(self):
        if self.wire_type == 'straight':
            self.create_straight_wire()
        elif self.wire_type == 'elbow':
            self.create_elbow_wire()
        elif self.wire_type == 'segment':
            self.create_segment_wire()

    def create_straight_wire(self):
        self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
width=self.width)

    def create_elbow_wire(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        if abs(w) >= abs(h): # Horizontal
            self.id = self.canvas.create_line(self.x1, self.y1, self.x2,
self.y1,
                                            self.x2, self.y1, self.x2,
self.y2,
                                            fill=self.fill_color,
                                            width=self.border_width,
tags="wire")
        else: # Vertical
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1,
self.y2,
                                            self.x1, self.y2, self.x2,
self.y2,
                                            fill=self.fill_color,
                                            width=self.border_width,
tags="wire")

    def create_segment_wire(self):
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        if abs(w) >= abs(h): # Horizontal
            self.seg1 = self.x1, self.y1, self.x1 + w / 2, self.y1
            self.seg2 = self.x1 + w / 2, self.y1, self.x1 + w / 2, self.y2
            self.seg3 = self.x1 + w / 2, self.y2, self.x2, self.y2
        else: # Vertical
            self.seg1 = self.x1, self.y1, self.x1, self.y1 + h / 2

```

```

        self(seg2 = self.x1, self.y1 + h / 2, self.x2, self.y1 + h / 2
        self(seg3 = self.x2, self.y1 + h / 2, self.x2, self.y2
    self.segment_list = [self.seg1, self.seg2, self.seg3]
    self.draw_segments()

def draw_segments(self):
    for s in self.segment_list:
        self.id = self.canvas.create_line(s, fill=self.fill_color,
width=self.border_width, tags='wire')

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_selectors()
    self.update_selection()

def update_position(self):
    if self.wire_type == 'straight':
        self.update_straight_position()
    elif self.wire_type == 'elbow':
        self.update_elbow_position()
    elif self.wire_type == 'segment':
        self.update_segment_position()

def update_straight_position(self):
    """Update the position when the attached component is moved"""
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)

def update_elbow_position(self):
    """Update the position when the attached component is moved"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    if abs(w) >= abs(h):
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y1,
                           self.x2, self.y1, self.x2, self.y2)
    else:
        self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y2,
                           self.x1, self.y2, self.x2, self.y2)

def update_segment_position(self):
    """Update the position when the attached component is moved"""
    w = self.x2 - self.x1
    h = self.y2 - self.y1
    if abs(w) >= abs(h):
        self.canvas.coords(self.id, self.x1, self.y1, self.x1 + w / 2,
self.y1,

```

```

        self.x1 + w / 2, self.y1, self.x1 + w / 2,
self.y2,
        self.x1 + w / 2, self.y2, self.x2, self.y2)
    else:
        self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y1 + h
/ 2,
                           self.x1, self.y1 + h / 2, self.x2, self.y1 + h /
2,
                           self.x2, self.y1 + h / 2, self.x2, self.y2)

def hit_test(self, x, y):
    # 2-Point Line equation: y = m * (x - x1) + y1
    x1, y1 = self.x1, self.y1
    x2, y2 = self.x2, self.y2

    # Calculate the slope: m = (y2 - y1) / (x2 - x1)
    if (x2 - x1) == 0:
        m = 0
    else:
        m = (y2 - y1)/(x2 - x1)

    # Check to see if the point (x, y) is on the line and between the two
end points
    tol = 10
    if y - tol <= m*(x - x1) + y1 <= y + tol:
        if (min(x1, x2) <= x <= max(x1, x2)) and (min(y1, y2) <= y <=
max(y1, y2)):
            self.is_selected = True
        else:
            self.is_selected = False

    def reprJson(self):
        return dict(type=self.wire_type, x1=self.x1, y1=self.y1, x2=self.x2,
y2=self.y2, name=self.name)

```

UI_Lib/wire_button_frame.py

```

import customtkinter as ctk
from pathlib import Path
from PIL import Image

from Wire_Lib import AnalogWire

```



```

        (Path(__file__).parent /
"../icons/elbow_line.png"),
size=(24, 24))

elbow_wire_button = ctk.CTkButton(self, text="",
image=elbow_wire_image, width=30,
command=self.create_elbow_wire)
elbow_wire_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)

# Shape button handlers
def create_straight_wire(self):
    wire = AnalogWire(self.canvas, 'straight', 0, 0, 0, 0)
    self.create_wire(wire)

def create_segment_wire(self):
    wire = AnalogWire(self.canvas, 'segment', 0, 0, 0, 0)
    self.create_wire(wire)

def create_elbow_wire(self):
    wire = AnalogWire(self.canvas, 'elbow', 0, 0, 0, 0)
    self.create_wire(wire)

def create_wire(self, wire):
    self.assign_wire_name(wire)
    self.canvas.mouse.current_wire_obj = wire
    self.canvas.show_connectors()
    self.canvas.comp_list.append(wire)
    self.canvas.mouse.draw_wire_mouse_events()

def assign_wire_name(self, wire):
    self.wire_count += 1
    wire_name = 'wire' + str(self.wire_count)
    wire.name = wire_name
    self.canvas.wire_dict[wire_name] = wire

```

UI.Lib/mouse.py

```

from Helper_Lib import Point
from Wire_Lib import AnalogWire
from Comp_Lib import Connection

class Mouse:

```

```

def __init__(self, canvas):
    self.canvas = canvas

    self.selected_comp = None
    self.current_wire_obj = None
    self.start = Point(0, 0)
    self.offset1 = Point(0, 0)
    self.offset2 = Point(0, 0)

    self.move_mouse_bind_events()

def move_mouse_bind_events(self):
    self.canvas.bind("<Button-1>", self.move_left_down)
    self.canvas.bind("<B1-Motion>", self.move_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

def draw_wire_mouse_events(self): # Added method to bind draw wire methods
    self.canvas.bind("<Button-1>", self.draw_left_down)
    self.canvas.bind("<B1-Motion>", self.draw_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

def resize_wire_mouse_events(self):
    self.canvas.bind("<Button-1>", self.resize_left_down)
    self.canvas.bind("<B1-Motion>", self.resize_left_drag)
    self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

def move_left_down(self, event):
    x, y = event.x, event.y
    self.comp_hit_test(x, y)
    if self.selected_comp:
        if isinstance(self.selected_comp, AnalogWire):
            x1, y1 = self.selected_comp.x1, self.selected_comp.y1
            x2, y2 = self.selected_comp.x2, self.selected_comp.y2
            self.offset1.set(x - x1, y - y1)
            self.offset2.set(x - x2, y - y2)
        else:
            x1, y1 = self.selected_comp.x1, self.selected_comp.y1
            self.offset1.x, self.offset1.y =
    self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y)
    self.offset1.set(x - x1, y - y1)

def move_left_drag(self, event):
    if self.selected_comp:
        if isinstance(self.selected_comp, AnalogWire):
            x1 = event.x - self.offset1.x
            y1 = event.y - self.offset1.y

```

```

        x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
        x2 = event.x - self.offset2.x
        y2 = event.y - self.offset2.y
        x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
        self.selected_comp.x1, self.selected_comp.y1 = x1, y1
        self.selected_comp.x2, self.selected_comp.y2 = x2, y2
        self.canvas.redraw()

    else:
        x = event.x - self.offset1.x
        y = event.y - self.offset1.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        self.selected_comp.x1, self.selected_comp.y1 = x, y
        self.canvas.redraw()

def move_left_up(self, _event):
    self.offset1.set(0, 0)
    self.offset2.set(0, 0)
    self.canvas.redraw()

def draw_left_down(self, event): # Added method for draw left down
    if self.current_wire_obj:
        # self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y
        self.start.x, self.start.y =
self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

        self.current_wire_obj.x1, self.current_wire_obj.y1 = self.start.x,
self.start.y
        self.current_wire_obj.x2, self.current_wire_obj.y2 = self.start.x,
self.start.y

        self.select_connector(self.current_wire_obj, "begin", self.start.x,
self.start.y)

def draw_left_drag(self, event): # Added method for draw left drag
    if self.current_wire_obj:
        wire = self.current_wire_obj
        x, y = event.x, event.y
        x, y = self.canvas.grid.snap_to_grid(x, y)
        wire.x1, wire.y1 = self.start.x, self.start.y
        wire.x2, wire.y2 = x, y
        self.current_wire_obj.update()

def draw_left_up(self, event): # Added method for draw left up
    self.select_connector(self.current_wire_obj, "end", event.x, event.y)

```

```

        self.canvas.hide_connectors()
        self.current_wire_obj = None
        self.move_mouse_bind_events()

    def resize_left_down(self, event):
        if self.selected_comp:
            x1, y1 = self.selected_comp.x1, self.selected_comp.y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1, "resize")
            x2, y2 = self.selected_comp.x2, self.selected_comp.y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2, "resize")
            self.offset1.x = event.x - x1
            self.offset1.y = event.y - y1
            self.offset2.x = event.x - x2
            self.offset2.y = event.y - y2
            self.selected_comp.update()

    def resize_left_drag(self, event):
        if self.selected_comp:
            offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
self.offset2.y]
            self.selected_comp.resize(offsets, event)
            self.selected_comp.update()

    def resize_left_up(self, _event):
        self.offset1.x, self.offset1.y = 0, 0
        self.offset2.x, self.offset2.y = 0, 0
        self.move_mouse_bind_events()

    def comp_hit_test(self, x, y):
        for comp in self.canvas.comp_list:
            comp.hit_test(x, y)
            if comp.is_selected:
                if isinstance(self.selected_comp, AnalogWire):
                    result = comp.sel_hit_test(x, y)
                    if result is not None:
                        self.resize_wire_mouse_events()
                comp.update()
                self.selected_comp = comp
                return

        # No shape hit - unselect all
        self.selected_comp = None
        self.unselect_all()

    def unselect_all(self):
        for comp in self.canvas.comp_list:

```

```

        comp.is_selected = False
        comp.update()

    def select_connector(self, wire_obj, wire_end, x, y):
        for comp in self.canvas.comp_list:
            if not isinstance(comp, AnalogWire):
                conn = comp.check_connector_hit(x, y)
            if conn:
                if wire_end == "begin":
                    wire_obj.x1, wire_obj.y1 = conn.x, conn.y
                elif wire_end == "end":
                    wire_obj.x2, wire_obj.y2 = conn.x, conn.y
                a_conn = Connection(conn.name, self.current_wire_obj.name,
wire_end)
                wire_obj.create_wire_list_cnx(comp.comp_type, conn.name)
                comp.wire_list.append(a_conn)
                self.canvas.redraw()

```

Transistor Amplifier Components

Lets implement the AC Coupled Transistor Amplifier from [PySpice Evaluation](#)

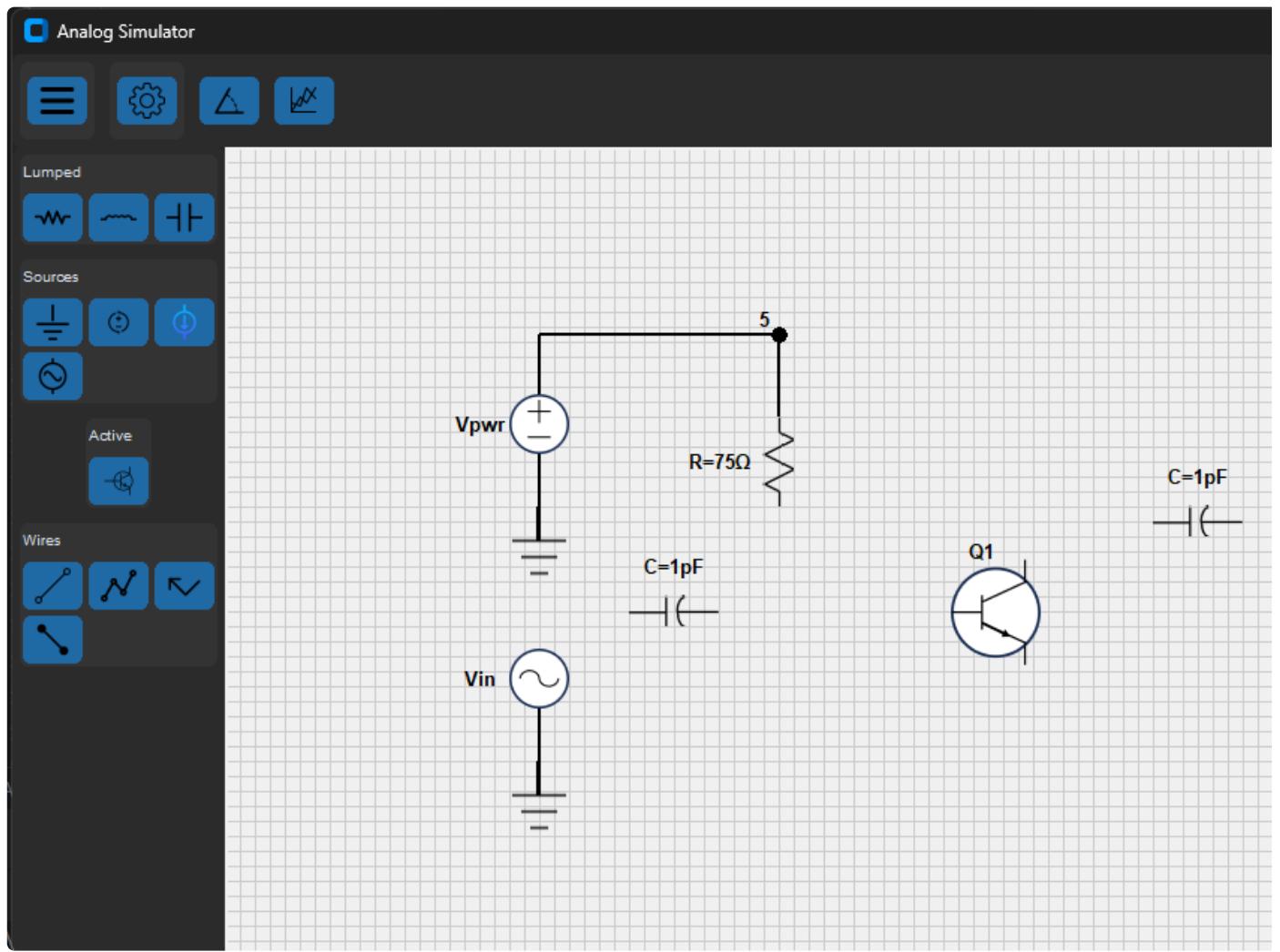
Components needed:

- DC Power Source
- AC Power Source
- Capacitor
- Resistor
- NPN Transistor
- Ground
- Nodes with Text - in, out, 0, 2, 3, 4, 5
- Matplotlib oscilloscope

PySpice Netlist

```
circuit.R(1, 5, 2, 100@u_kΩ)
```

- 1 = Resistor label as in R1
- 5, 2 = input and output nodes
- 100@u_kΩ = resistance with units



Comp_Lib/component.py

```

import tkinter as tk
from PIL import Image, ImageTk
from pathlib import Path

from Helper_Lib import Point


class Component:
    def __init__(self, canvas, comp_type, x1, y1, value):
        """Base class for component classes"""
        self.canvas = canvas
        self.comp_type = comp_type
        self.x1 = x1
        self.y1 = y1
        self.value = value

        self.id = None

```

```

self.sel_id = None

self.is_selected = False
self.is_drawing = False
self.selector = None
self.angle = 0
self.comp_text = None
self.text = None
self.text_id = None

self.filename = None
self.a_image = None
self.ph_image = None
self.bbox = None

self.out = None
self.in1 = None
self.ba = None
self.em = None
self.co = None

self.conn_list = []
self.wire_list = []

# Define component parameters in a dictionary of dictionaries
self.params = {
    'filename': {
        'resistor': '../images/lumped/resistor_60x30.png',
        'capacitor': '../images/lumped/capacitor_60x30.png',
        'inductor': '../images/lumped/inductor_60x30.png',
        'ground': '../images/sources/ground_50x40.png',
        'vsource': '../images/sources/voltage_source_40x40.png',
        'isource': '../images/sources/current_source_40x40.png',
        'ac_source': '../images/sources/ac_voltage_source_40x40.png',
        'npn_transistor':
            '../images/transistors/npn_transistor_60x71.png'
    },
    'text': {
        'resistor': 'R=' + str(self.value) + '\u02126',
        'capacitor': 'C=' + str(self.value) + 'pF',
        'inductor': 'L=' + str(self.value) + 'nH',
        'ground': '',
        'vsource': 'Vpwr',
        'isource': 'I',
        'ac_source': 'Vin',
        'npn_transistor': 'Q1'
    }
}

```

```

        }

    }

    def set_image_filename(self):
        self.filename = Path(__file__).parent / self.params['filename']
        [self.comp_type]

    def create_text(self):
        self.comp_text = self.params['text'][self.comp_type]
        if self.comp_type == 'isource' or self.comp_type == 'vsource' or
        self.comp_type == 'ac_source':
            text_loc = Point(self.x1-40, self.y1) # Put text on left side of
        symbol
        elif self.comp_type == 'npn_transistor':
            text_loc = Point(self.x1 - 10, self.y1 - 40) # Put text above
        symbol
        else:
            text_loc = Point(self.x1, self.y1-30) # Put text above symbol
        self.text_id = self.canvas.create_text(text_loc.x, text_loc.y,
                                              text=self.comp_text, fill="black",
                                              font='Helvetica 10 bold',
                                              angle=self.angle, tags="text")

    def create_image(self, filename):
        """Initial component image creation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
                                         image=self.ph_image, tags='gate')

    def update_position(self):
        """Update the position when the gate object is moved"""
        self.canvas.coords(self.id, self.x1, self.y1) # Update position

    def update_image(self, filename):
        """Update the image for gate symbol rotation"""
        self.a_image = Image.open(filename)
        self.a_image = self.a_image.rotate(self.angle, expand=True) # Update
        image rotation
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.canvas.itemconfig(self.id, image=self.ph_image) # Update image

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

```

```

def update_text(self):
    if self.comp_type == 'isource' or self.comp_type == 'vsource' or
self.comp_type == 'ac_source':
        if self.angle == 0 or self.angle == 180:
            self.set_east_text()
        elif self.angle == 90 or self.angle == 270:
            self.set_north_text()
    elif self.comp_type == 'npn_transistor':
        self.set_transistor_text()
    else:
        if self.angle == 0 or self.angle == 180:
            self.set_north_text()
        elif self.angle == 90 or self.angle == 270:
            self.set_east_text()

def set_east_text(self):
    self.canvas.coords(self.text_id, self.x1 - 40, self.y1)

def set_north_text(self):
    self.canvas.coords(self.text_id, self.x1, self.y1 - 30)

def set_transistor_text(self):
    self.canvas.coords(self.text_id, self.x1 - 10, self.y1 - 40)

def create_selector(self):
    """Create the red rectangle selector and check to see if the gate is
selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
    self.set_selector_visibility()

def update_selector(self):
    """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.canvas.coords(self.sel_id, x1, y1, x2, y2)
    self.set_selector_visibility()

def set_selector_visibility(self):
    """Set the selector visibility state"""
    if self.is_selected:
        self.canvas.itemconfig(self.sel_id, state='normal')

```

```

        else:
            self.canvas.itemconfig(self.sel_id, state='hidden')

    def set_connector_visibility(self):
        """Set the connector visibility state"""
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

    def check_connector_hit(self, x, y):
        """Hit test to see if a connector is at the provided x, y
coordinates"""
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None

    def move_connected_wires(self):
        """Resize connected wires if the shape is moved"""
        for connection in self.wire_list: # comp_conn, wire_name, wire_end
            for connector in self.conn_list:
                if connector.name == connection.comp_conn:
                    wire_obj = self.canvas.wire_dict[connection.wire_name]
                    if connection.wire_end == "begin":
                        wire_obj.x1 = connector.x
                        wire_obj.y1 = connector.y
                    elif connection.wire_end == "end":
                        wire_obj.x2 = connector.x
                        wire_obj.y2 = connector.y

    def rotate(self):
        """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
if angle > 270 deg"""
        self.angle += 90
        if self.angle > 270:
            self.angle = 0

    def hit_test(self, x, y):
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        if x1 <= x <= x2 and y1 <= y <= y2:
            self.is_selected = True
        else:

```

```
    self.is_selected = False
```

Comp_Lib/analog_component.py

```
from Comp_Lib.component import Component
from Comp_Lib.connector import Connector
from Helper_Lib import Point

class AnalogComponent(Component):
    """Universal model for analog components"""
    def __init__(self, canvas, comp_type, x1, y1, value=0):
        super().__init__(canvas, comp_type, x1, y1, value)

        self.conn_params = {
            'ports': {
                'resistor': 2,
                'capacitor': 2,
                'inductor': 2,
                'ground': 1,
                'vsource': 2,
                'isource': 2,
                'ac_source': 2,
                'npn_transistor': 3
            },
            'conn_loc': {
                'resistor': 'ew',
                'capacitor': 'ew',
                'inductor': 'ew',
                'ground': 'n',
                'vsource': 'ns',
                'isource': 'ns',
                'ac_source': 'ns',
                'npn_transistor': 'nsw'
            }
        }

        self.create()

    def create(self):
        self.set_image_filename()
        self.create_image(self.filename)
        self.update_bbox()
```

```

        self.create_text()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def update(self):
        self.update_position()
        self.update_image(self.filename)
        self.update_bbox()
        self.update_text()
        self.update_selector()
        self.update_connectors()
        self.set_connector_visibility()

    def create_connectors(self):
        # Calculate position of connectors from current comp position and size
        center, e, w, n, s = self.get_geometry()

        num_ports = self.conn_params['ports'][self.comp_type]
        if num_ports == 1:
            self.out = Connector(self.canvas, "in1", center.x, center.y)
            self.conn_list = [self.out]
        elif num_ports == 2:
            self.out = Connector(self.canvas, "out", center.x, center.y)
            self.in1 = Connector(self.canvas, "in1", center.x, center.y)
            self.conn_list = [self.in1, self.out]
        elif num_ports == 3:
            self.ba = Connector(self.canvas, "base", center.x, center.y)
            self.em = Connector(self.canvas, "emitter", center.x, center.y)
            self.co = Connector(self.canvas, "collector", center.x, center.y)
            self.conn_list = [self.ba, self.em, self.co]

    def update_connectors(self):
        """Update the position of connectors here"""
        center, e, w, n, s = self.get_geometry()

        conn_loc = self.conn_params['conn_loc'][self.comp_type]
        if conn_loc == 'ew': # 2-port with ew ports
            self.calc_ew_conn_rotation(n, s, e, w)
        elif conn_loc == 'ns': # 2-port with ns ports
            self.calc_ns_conn_rotation(n, s, e, w)
        elif conn_loc == 'n': # 1-port with n port
            self.calc_n_conn_rotation(n, w)
        elif conn_loc == 'nsw': # 3-port with nsw ports
            self.calc_nsw_conn_rotation(n, s, e, w)

```

```

        for c in self.conn_list:
            c.update()

        self.move_connected_wires()

    def calc_ew_conn_rotation(self, n, s, e, w):
        if self.angle == 0 or self.angle == 180:
            self.out.x, self.out.y = w.x, w.y
            self.in1.x, self.in1.y = e.x, e.y
        elif self.angle == 90 or self.angle == 270:
            self.out.x, self.out.y = n.x, n.y
            self.in1.x, self.in1.y = s.x, s.y

    def calc_ns_conn_rotation(self, n, s, e, w):
        if self.angle == 0 or self.angle == 180:
            self.out.x, self.out.y = n.x, n.y
            self.in1.x, self.in1.y = s.x, s.y
        elif self.angle == 90 or self.angle == 270:
            self.out.x, self.out.y = w.x, w.y
            self.in1.x, self.in1.y = e.x, e.y

    def calc_n_conn_rotation(self, n, w):
        if self.angle == 0 or self.angle == 180:
            self.out.x, self.out.y = n.x, n.y
        elif self.angle == 90 or self.angle == 270:
            self.out.x, self.out.y = w.x, w.y

    def calc_nsw_conn_rotation(self, n, s, e, w):
        if self.angle == 0 or self.angle == 180:
            self.em.x, self.em.y = n.x, n.y
            self.co.x, self.co.y = s.x, s.y
            self.ba.x, self.ba.y = w.x, w.y
        elif self.angle == 90 or self.angle == 270:
            self.em.x, self.em.y = w.x, w.y
            self.co.x, self.co.y = e.x, e.y
            self.ba.x, self.ba.y = s.x, s.y

    def get_geometry(self):
        sign = lambda angle: 1 if angle == 0 or angle == 180 else -1

        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.comp_type == 'nnp_transistor':

```

```

        e = Point(center.x + sign(self.angle) * w / 2, center.y)
        w = Point(center.x - sign(self.angle) * w / 2, center.y)
        n = Point(center.x + 20, center.y - sign(self.angle) * h / 2)
        s = Point(center.x + 20, center.y + sign(self.angle) * h / 2)
    else:
        e = Point(center.x - sign(self.angle) * w / 2, center.y)
        w = Point(center.x + sign(self.angle) * w / 2, center.y)
        n = Point(center.x, center.y - sign(self.angle) * h / 2)
        s = Point(center.x, center.y + sign(self.angle) * h / 2)

    return center, e, w, n, s

def __repr__(self):
    return ("Type: " + self.comp_type + " x1: " + str(self.x1) + " y1: " +
str(self.y1) + " value: " +
            str(self.value) + " wire list: " +
str(self.wire_list.__repr__()))

def reprJson(self):
    return dict(type=self.comp_type, x1=self.x1, y1=self.y1,
angle=self.angle, value=self.value,
            wire_list=self.wire_list)

```

Wire_Lib/node.py

```

from Helper_Lib import Point
from Comp_Lib import Connector

class Node:
    def __init__(self, canvas, x1, y1):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.text = '0'
        self.comp_type = 'node'
        self.id = None
        self.text_id = None
        self.sel_id = None
        self.wire_list = []
        self.bbox = None
        self.angle = 0

```

```

        self.is_selected = False
        self.is_drawing = False
        self.conn = None
        self.conn_list = []

    self.create()

def create(self):
    self.create_node()
    self.update_bbox()
    self.create_text()
    self.create_selector()
    self.create_connectors()
    self.set_connector_visibility()

def create_node(self):
    self.id = self.canvas.create_oval(self.x1 - 5, self.y1 - 5, self.x1 +
5, self.y1 + 5, fill="black")

def update_bbox(self):
    """Update the bounding box to get current gate coordinates"""
    self.bbox = self.canvas.bbox(self.id)

def create_text(self):
    text_loc = Point(self.x1-10, self.y1-10) # Put text above symbol
    self.text_id = self.canvas.create_text(text_loc.x, text_loc.y,
                                         text=self.text, fill="black",
                                         font='Helvetica 10 bold',
                                         angle=self.angle, tags="text")

def create_selector(self):
    """Create the red rectangle selector and check to see if the gate is
selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
    self.set_selector_visibility()

def set_selector_visibility(self):
    """Set the selector visibility state"""
    if self.is_selected:
        self.canvas.itemconfig(self.sel_id, state='normal')
    else:
        self.canvas.itemconfig(self.sel_id, state='hidden')

```

```

def create_connectors(self):
    # Calculate position of connectors from current comp position and size
    center = self.get_geometry()

    self.conn = Connector(self.canvas, "conn", center.x, center.y)
    self.conn_list = [self.conn]

def get_geometry(self):
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    w = x2 - x1
    h = y2 - y1
    center = Point(x1 + w / 2, y1 + h / 2)

    return center

def set_connector_visibility(self):
    """Set the connector visibility state"""
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_text()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def update_position(self):
    self.canvas.coords(self.id, self.x1 - 5, self.y1 - 5, self.x1 + 5,
self.y1 + 5) # Update position

def update_text(self):
    self.canvas.itemconfig(self.text_id, text=self.text)
    self.canvas.coords(self.text_id, self.x1 - 10, self.y1 - 10)

def update_selector(self):
    """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.canvas.coords(self.sel_id, x1, y1, x2, y2)

```

```

    self.set_selector_visibility()

def update_connectors(self):
    """Update the position of connectors here"""
    center = self.get_geometry()

    self.conn.x, self.conn.y = center.x, center.y

    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def check_connector_hit(self, x, y):
    """Hit test to see if a connector is at the provided x, y
    coordinates"""
    for conn in self.conn_list:
        if conn.connector_hit_test(x, y):
            return conn
    return None

def move_connected_wires(self):
    """Resize connected wires if the shape is moved"""
    for connection in self.wire_list: # comp_conn, wire_name, wire_end
        for connector in self.conn_list:
            if connector.name == connection.comp_conn:
                wire_obj = self.canvas.wire_dict[connection.wire_name]
                if connection.wire_end == "begin":
                    wire_obj.x1 = connector.x
                    wire_obj.y1 = connector.y
                elif connection.wire_end == "end":
                    wire_obj.x2 = connector.x
                    wire_obj.y2 = connector.y

def rotate(self):
    """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
    if angle > 270 deg"""
    self.angle += 90
    if self.angle > 270:
        self.angle = 0

def hit_test(self, x, y):
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    if x1 <= x <= x2 and y1 <= y <= y2:
        self.is_selected = True
    else:

```

```
    self.is_selected = False
```

UI_Lib/canvas.py

```
import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid
from Wire_Lib import Node


class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)
        self.led_color = "red"
        self.led_size = "large"
        self.grid = Grid(self, 10)
        self.mouse = Mouse(self)
        self.comp_list = []
        self.wire_list = []
        self.conn_list = []
        self.wire_dict = {}

        self.mouse.move_mouse_bind_events()

    def redraw(self):
        self.delete('grid_line')
        self.grid.draw()
        self.tag_lower("grid_line")
        for c in self.comp_list:
            c.update()

    def show_connectors(self):
        for s in self.comp_list:
            s.is_drawing = True
        self.redraw()

    def hide_connectors(self):
        for s in self.comp_list:
            s.is_drawing = False
        self.redraw()

    def edit_shape(self, _event=None):
```

```

    if self.mouse.selected_comp:
        comp = self.mouse.selected_comp
        if isinstance(comp, Node):
            dialog = ctk.CTkInputDialog(text="Enter new text", title="Edit
Node Text")
            comp.text = dialog.get_input()
            self.redraw()

```

analog_simulator.py

```

import customtkinter as ctk
from UI_Lib import LeftFrame, TopFrame, Canvas

ctk.set_appearance_mode("System") # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue") # Themes: "blue" (standard), "green",
"dark-blue"

class App(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Analog Simulator")

        self.canvas = Canvas(self)
        self.left_frame = LeftFrame(self, self.canvas)
        self.top_frame = TopFrame(self, self.canvas)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add bindings here
        self.bind("<Configure>", self.on_window_resize)
        self.bind('<r>', self.rotate_comp)
        self.canvas.bind('<Button-3>', self.canvas.edit_shape)

    def on_window_resize(self, _event):
        self.canvas.redraw()

```

```

def rotate_comp(self, _event=None):
    if self.canvas.mouse.selected_comp:
        self.canvas.mouse.selected_comp.rotate()
        self.canvas.redraw()

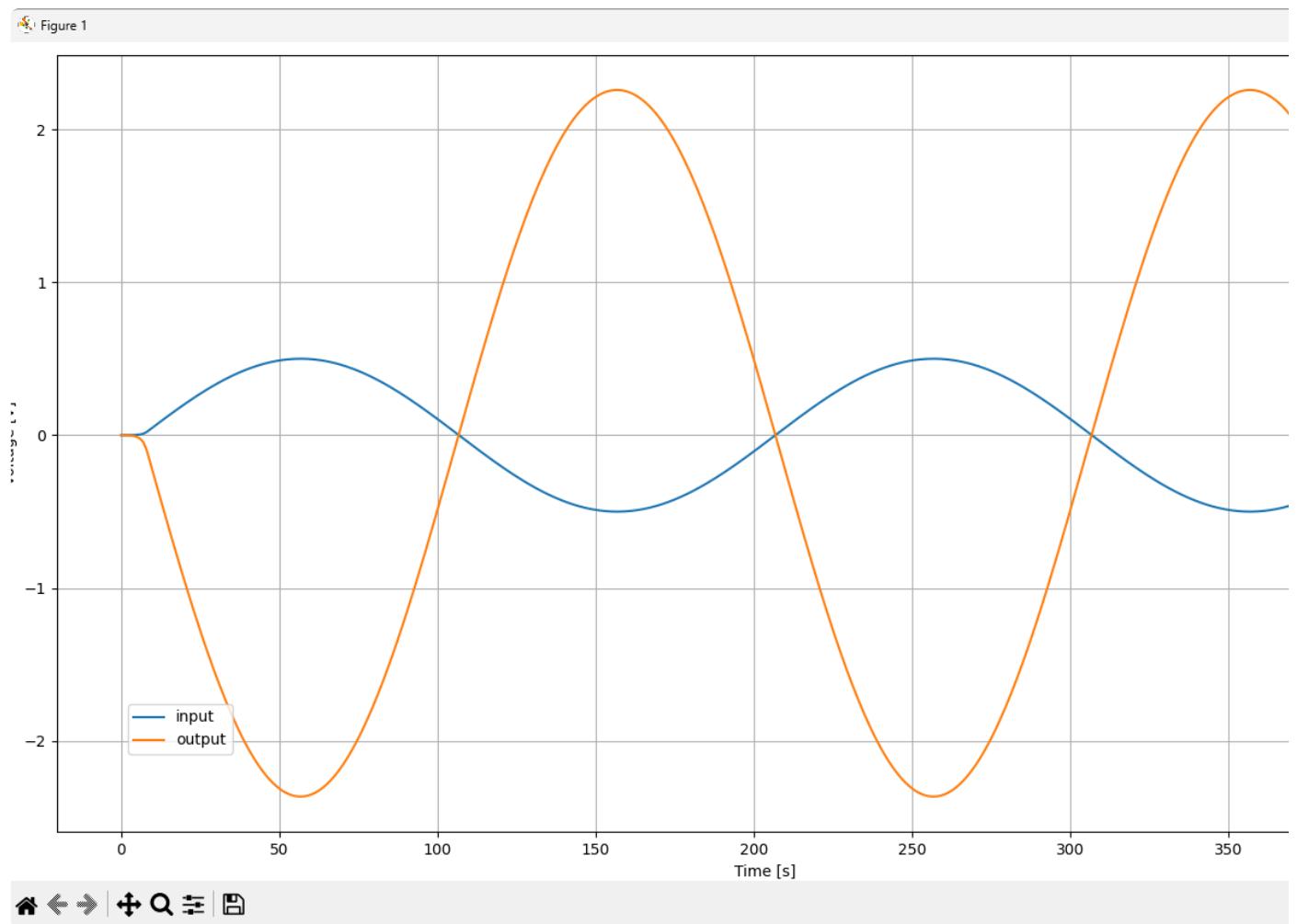
if __name__ == "__main__":
    """Instantiate the Microwave Simulator app and run the main loop"""
    app = App()
    app.mainloop()

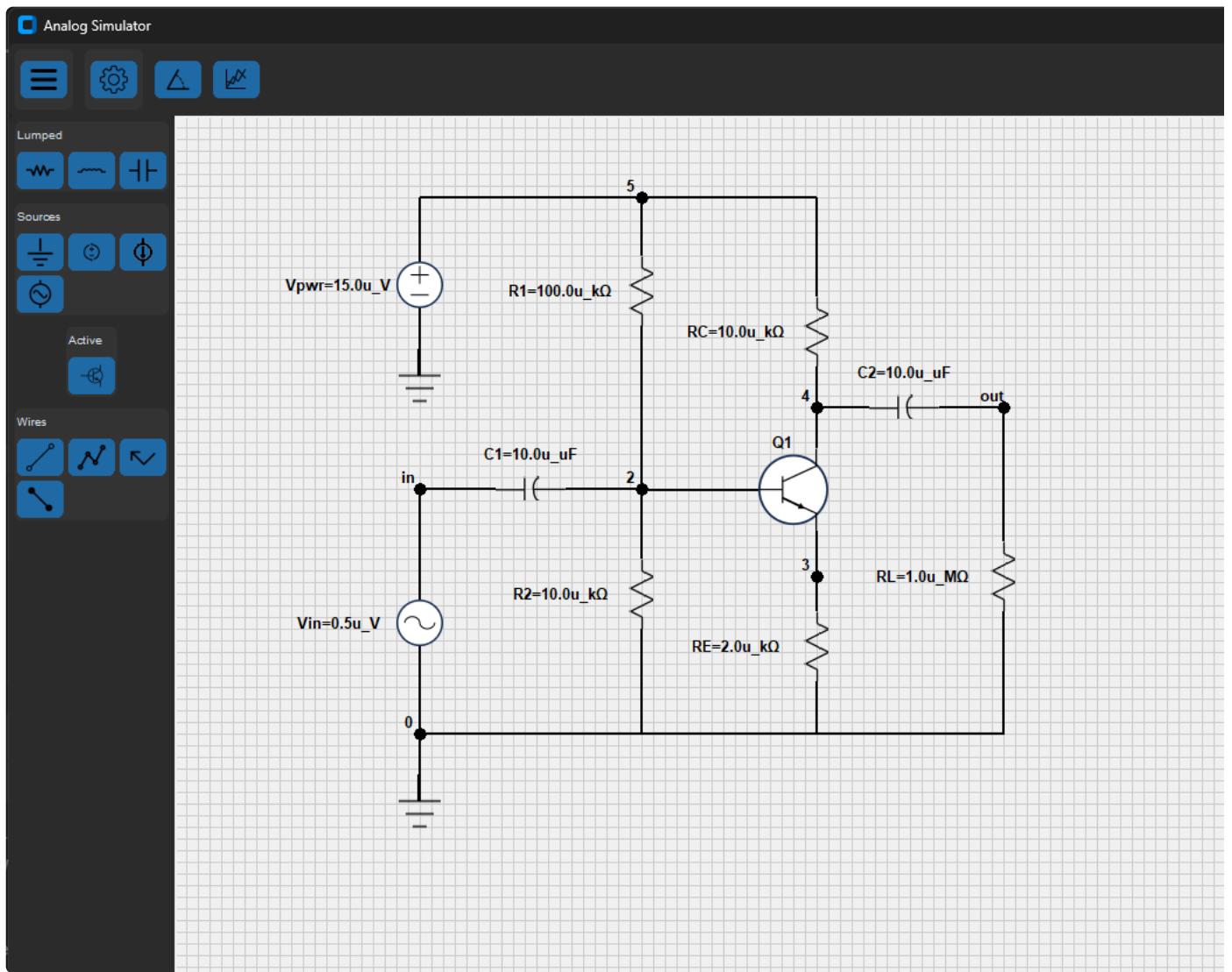
```

Transistor Amplifier Circuit

Objective:

- Convert Canvas Component List to PySpice netlist
- Run Analysis





Circuits/transistor-amplifier.json

```
[
  {
    "comp_list": [
      {
        "type": "dc_source",
        "text": "Vpwr",
        "x1": 210,
        "y1": 145,
        "angle": 0,
        "value": "15",
        "units": "u_V",
        "wire_list": [
          {
            "type": "connection",
            "comp_conn": "out",
            "x2": 210,
            "y2": 100
          }
        ]
      },
      {
        "type": "voltage_source",
        "text": "Vin",
        "x1": 100,
        "y1": 145,
        "angle": 90,
        "value": "0.5",
        "units": "u_V",
        "wire_list": [
          {
            "type": "connection",
            "comp_conn": "in",
            "x2": 100,
            "y2": 100
          }
        ]
      },
      {
        "type": "resistor",
        "text": "R1=100.0u_kΩ",
        "x1": 350,
        "y1": 145,
        "x2": 350,
        "y2": 100
      },
      {
        "type": "resistor",
        "text": "R2=10.0u_kΩ",
        "x1": 350,
        "y1": 300,
        "x2": 350,
        "y2": 250
      },
      {
        "type": "resistor",
        "text": "RC=10.0u_kΩ",
        "x1": 650,
        "y1": 145,
        "x2": 650,
        "y2": 100
      },
      {
        "type": "resistor",
        "text": "RE=2.0u_kΩ",
        "x1": 650,
        "y1": 300,
        "x2": 650,
        "y2": 250
      },
      {
        "type": "resistor",
        "text": "RL=1.0u_MΩ",
        "x1": 750,
        "y1": 300,
        "x2": 750,
        "y2": 250
      },
      {
        "type": "capacitor",
        "text": "C1=10.0u_uF",
        "x1": 350,
        "y1": 250,
        "x2": 350,
        "y2": 100
      },
      {
        "type": "capacitor",
        "text": "C2=10.0u_uF",
        "x1": 750,
        "y1": 250,
        "x2": 750,
        "y2": 100
      }
    ],
    "node_list": [
      {
        "name": "in",
        "x": 100,
        "y": 145
      },
      {
        "name": "out",
        "x": 750,
        "y": 100
      },
      {
        "name": "0",
        "x": 100,
        "y": 100
      },
      {
        "name": "2",
        "x": 350,
        "y": 145
      },
      {
        "name": "3",
        "x": 650,
        "y": 250
      },
      {
        "name": "4",
        "x": 750,
        "y": 250
      },
      {
        "name": "5",
        "x": 350,
        "y": 145
      }
    ],
    "connection_list": [
      {
        "x1": 210,
        "y1": 145,
        "x2": 350,
        "y2": 145
      },
      {
        "x1": 100,
        "y1": 145,
        "x2": 100,
        "y2": 100
      },
      {
        "x1": 100,
        "y1": 100,
        "x2": 350,
        "y2": 100
      },
      {
        "x1": 350,
        "y1": 100,
        "x2": 350,
        "y2": 145
      },
      {
        "x1": 350,
        "y1": 145,
        "x2": 650,
        "y2": 145
      },
      {
        "x1": 650,
        "y1": 145,
        "x2": 650,
        "y2": 250
      },
      {
        "x1": 650,
        "y1": 250,
        "x2": 750,
        "y2": 250
      },
      {
        "x1": 750,
        "y1": 250,
        "x2": 750,
        "y2": 100
      },
      {
        "x1": 750,
        "y1": 100,
        "x2": 350,
        "y2": 100
      },
      {
        "x1": 350,
        "y1": 100,
        "x2": 350,
        "y2": 250
      },
      {
        "x1": 350,
        "y1": 250,
        "x2": 100,
        "y2": 250
      },
      {
        "x1": 100,
        "y1": 250,
        "x2": 100,
        "y2": 145
      },
      {
        "x1": 100,
        "y1": 145,
        "x2": 210,
        "y2": 145
      },
      {
        "x1": 210,
        "y1": 145,
        "x2": 210,
        "y2": 100
      },
      {
        "x1": 210,
        "y1": 100,
        "x2": 350,
        "y2": 100
      },
      {
        "x1": 350,
        "y1": 100,
        "x2": 350,
        "y2": 145
      },
      {
        "x1": 350,
        "y1": 145,
        "x2": 650,
        "y2": 145
      },
      {
        "x1": 650,
        "y1": 145,
        "x2": 650,
        "y2": 250
      },
      {
        "x1": 650,
        "y1": 250,
        "x2": 750,
        "y2": 250
      },
      {
        "x1": 750,
        "y1": 250,
        "x2": 750,
        "y2": 100
      },
      {
        "x1": 750,
        "y1": 100,
        "x2": 350,
        "y2": 100
      },
      {
        "x1": 350,
        "y1": 100,
        "x2": 350,
        "y2": 250
      },
      {
        "x1": 350,
        "y1": 250,
        "x2": 100,
        "y2": 250
      },
      {
        "x1": 100,
        "y1": 250,
        "x2": 100,
        "y2": 145
      },
      {
        "x1": 100,
        "y1": 145,
        "x2": 210,
        "y2": 145
      }
    ]
  }
]
```

```
        "wire_name": "wire1",
        "wire_end": "begin"
    },
    {
        "type": "connection",
        "comp_conn": "in1",
        "wire_name": "wire2",
        "wire_end": "begin"
    }
]
},
{
    "type": "resistor",
    "text": "R1",
    "x1": 400,
    "y1": 150,
    "angle": 90,
    "value": "100",
    "units": "u_k\u03a9",
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire7",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire8",
            "wire_end": "begin"
        }
    ]
},
{
    "type": "npn_transistor",
    "text": "Q1",
    "x1": 530,
    "y1": 320,
    "angle": 0,
    "bf": 80,
    "cjc": 5,
    "cjc_units": "u_pF",
    "rb": 100,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire9",
            "wire_end": "begin"
        }
    ]
}
]
```

```
        "type": "connection",
        "comp_conn": "base",
        "wire_name": "wire10",
        "wire_end": "end"
    },
    {
        "type": "connection",
        "comp_conn": "emitter",
        "wire_name": "wire14",
        "wire_end": "end"
    },
    {
        "type": "connection",
        "comp_conn": "collector",
        "wire_name": "wire18",
        "wire_end": "begin"
    }
]
},
{
    "type": "resistor",
    "text": "RC",
    "x1": 550,
    "y1": 185,
    "angle": 90,
    "value": "10",
    "units": "u_k\u03a9",
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire12",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire13",
            "wire_end": "begin"
        }
    ]
},
{
    "type": "node",
    "text": "5",
    "x1": 400,
```

```
        "y1": 70,
        "angle": 0,
        "wire_list": [
            {
                "type": "connection",
                "comp_conn": "conn",
                "wire_name": "wire1",
                "wire_end": "end"
            },
            {
                "type": "connection",
                "comp_conn": "conn",
                "wire_name": "wire7",
                "wire_end": "begin"
            },
            {
                "type": "connection",
                "comp_conn": "conn",
                "wire_name": "wire12",
                "wire_end": "begin"
            }
        ]
    },
    {
        "type": "node",
        "text": "4",
        "x1": 550,
        "y1": 250,
        "angle": 0,
        "wire_list": [
            {
                "type": "connection",
                "comp_conn": "conn",
                "wire_name": "wire13",
                "wire_end": "end"
            },
            {
                "type": "connection",
                "comp_conn": "conn",
                "wire_name": "wire14",
                "wire_end": "begin"
            },
            {
                "type": "connection",
                "comp_conn": "conn",
                "wire_name": "wire15",
                "wire_end": "begin"
            }
        ]
    }
]
```

```
        "wire_end": "begin"
    }
]
},
{
    "type": "node",
    "text": "3",
    "x1": 550,
    "y1": 395,
    "angle": 0,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire18",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire19",
            "wire_end": "begin"
        }
    ]
},
{
    "type": "node",
    "text": "2",
    "x1": 400,
    "y1": 320,
    "angle": 0,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire8",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire9",
            "wire_end": "end"
        },
        {
            "type": "connection",

```

```
        "comp_conn": "conn",
        "wire_name": "wire10",
        "wire_end": "begin"
    },
    {
        "type": "connection",
        "comp_conn": "conn",
        "wire_name": "wire11",
        "wire_end": "begin"
    }
]
},
{
    "type": "capacitor",
    "text": "C1",
    "x1": 305,
    "y1": 320,
    "angle": 0,
    "value": "10",
    "units": "u_uF",
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire3",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire9",
            "wire_end": "begin"
        }
    ]
},
{
    "type": "ac_source",
    "text": "Vin",
    "x1": 210,
    "y1": 435,
    "angle": 0,
    "value": "0.5",
    "units": "u_V",
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire10",
            "wire_end": "begin"
        }
    ]
}
]
```

```
        "comp_conn": "out",
        "wire_name": "wire4",
        "wire_end": "end"
    },
    {
        "type": "connection",
        "comp_conn": "in1",
        "wire_name": "wire5",
        "wire_end": "begin"
    }
]
},
{
    "type": "node",
    "text": "in",
    "x1": 210,
    "y1": 320,
    "angle": 0,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire3",
            "wire_end": "begin"
        },
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire4",
            "wire_end": "begin"
        }
    ]
},
{
    "type": "ground",
    "text": null,
    "x1": 210,
    "y1": 225,
    "angle": 0,
    "value": 0,
    "units": null,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire2",
            "wire_end": "begin"
        }
    ]
}
```

```
        "wire_end": "end"
    }
]
},
{
    "type": "ground",
    "text": null,
    "x1": 210,
    "y1": 590,
    "angle": 0,
    "value": 0,
    "units": null,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire6",
            "wire_end": "end"
        }
    ]
},
{
    "type": "node",
    "text": "0",
    "x1": 210,
    "y1": 530,
    "angle": 0,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire5",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire6",
            "wire_end": "begin"
        },
        {
            "type": "connection",
            "comp_conn": "conn",
            "wire_name": "wire20",
            "wire_end": "begin"
        }
    ]
}
```

```
{
    "type": "connection",
    "comp_conn": "conn",
    "wire_name": "wire21",
    "wire_end": "begin"
},
{
    "type": "connection",
    "comp_conn": "conn",
    "wire_name": "wire22",
    "wire_end": "begin"
}
]
},
{
    "type": "resistor",
    "text": "RE",
    "x1": 550,
    "y1": 455,
    "angle": 90,
    "value": "2",
    "units": "u_k\u00a9",
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire19",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire21",
            "wire_end": "end"
        }
    ]
},
{
    "type": "resistor",
    "text": "R2",
    "x1": 400,
    "y1": 410,
    "angle": 90,
    "value": "10",
    "units": "u_k\u00a9",
    "wire_list": [

```

```
{
    "type": "connection",
    "comp_conn": "in1",
    "wire_name": "wire11",
    "wire_end": "end"
},
{
    "type": "connection",
    "comp_conn": "out",
    "wire_name": "wire22",
    "wire_end": "end"
}
]
},
{
    "type": "capacitor",
    "text": "C2",
    "x1": 625,
    "y1": 250,
    "angle": 0,
    "value": "10",
    "units": "u_uF",
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire15",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire16",
            "wire_end": "begin"
        }
    ]
},
{
    "type": "node",
    "text": "out",
    "x1": 710,
    "y1": 250,
    "angle": 0,
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire17",
            "wire_end": "begin"
        }
    ]
}
```

```
        "comp_conn": "conn",
        "wire_name": "wire16",
        "wire_end": "end"
    },
    {
        "type": "connection",
        "comp_conn": "conn",
        "wire_name": "wire17",
        "wire_end": "begin"
    }
]
},
{
    "type": "resistor",
    "text": "RL",
    "x1": 710,
    "y1": 395,
    "angle": 90,
    "value": "1",
    "units": "\u03a9",
    "wire_list": [
        {
            "type": "connection",
            "comp_conn": "in1",
            "wire_name": "wire17",
            "wire_end": "end"
        },
        {
            "type": "connection",
            "comp_conn": "out",
            "wire_name": "wire20",
            "wire_end": "end"
        }
    ]
},
{
    "type": "elbow",
    "wire_dir": "V",
    "node_num": "5",
    "x1": 210.0,
    "y1": 125.0,
    "x2": 400.0,
    "y2": 70.0,
    "name": "wire1"
},
{
```

```
        "type": "straight",
        "wire_dir": "H",
        "node_num": "0",
        "x1": 210.0,
        "y1": 165.0,
        "x2": 210.0,
        "y2": 200.0,
        "name": "wire2"
    },
    {
        "type": "straight",
        "wire_dir": "H",
        "node_num": "in",
        "x1": 210.0,
        "y1": 320.0,
        "x2": 275.0,
        "y2": 320.0,
        "name": "wire3"
    },
    {
        "type": "straight",
        "wire_dir": "H",
        "node_num": "in",
        "x1": 210.0,
        "y1": 320.0,
        "x2": 210.0,
        "y2": 415.0,
        "name": "wire4"
    },
    {
        "type": "straight",
        "wire_dir": "H",
        "node_num": "0",
        "x1": 210.0,
        "y1": 455.0,
        "x2": 210.0,
        "y2": 530.0,
        "name": "wire5"
    },
    {
        "type": "straight",
        "wire_dir": "H",
        "node_num": "0",
        "x1": 210.0,
        "y1": 530.0,
        "x2": 210.0,
```

```
        "y2": 565.0,
        "name": "wire6"
    },
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "5",
    "x1": 400.0,
    "y1": 70.0,
    "x2": 400.0,
    "y2": 120.0,
    "name": "wire7"
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "2",
    "x1": 400.0,
    "y1": 180.0,
    "x2": 400.0,
    "y2": 320.0,
    "name": "wire8"
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "2",
    "x1": 335.0,
    "y1": 320.0,
    "x2": 400.0,
    "y2": 320.0,
    "name": "wire9"
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "2",
    "x1": 400.0,
    "y1": 320.0,
    "x2": 500.0,
    "y2": 320.5,
    "name": "wire10"
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "2",
    "x1": 500.0,
```

```
        "node_num": "2",
        "x1": 400.0,
        "y1": 320.0,
        "x2": 400.0,
        "y2": 380.0,
        "name": "wire11"
    },
    {
        "type": "elbow",
        "wire_dir": "H",
        "node_num": "5",
        "x1": 400.0,
        "y1": 70.0,
        "x2": 550.0,
        "y2": 155.0,
        "name": "wire12"
    },
    {
        "type": "straight",
        "wire_dir": "H",
        "node_num": "4",
        "x1": 550.0,
        "y1": 215.0,
        "x2": 550.0,
        "y2": 250.0,
        "name": "wire13"
    },
    {
        "type": "straight",
        "wire_dir": "H",
        "node_num": "4",
        "x1": 550.0,
        "y1": 250.0,
        "x2": 550.0,
        "y2": 285.0,
        "name": "wire14"
    },
    {
        "type": "straight",
        "wire_dir": "H",
        "node_num": "4",
        "x1": 550.0,
        "y1": 250.0,
        "x2": 595.0,
        "y2": 250.0,
        "name": "wire15"
    }
```

```
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "out",
    "x1": 655.0,
    "y1": 250.0,
    "x2": 710.0,
    "y2": 250.0,
    "name": "wire16"
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "out",
    "x1": 710.0,
    "y1": 250.0,
    "x2": 710.0,
    "y2": 365.0,
    "name": "wire17"
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "3",
    "x1": 550.0,
    "y1": 356.0,
    "x2": 550.0,
    "y2": 395.0,
    "name": "wire18"
},
{
    "type": "straight",
    "wire_dir": "H",
    "node_num": "3",
    "x1": 550.0,
    "y1": 395.0,
    "x2": 550.0,
    "y2": 425.0,
    "name": "wire19"
},
{
    "type": "elbow",
    "wire_dir": "H",
    "node_num": "0",
    "x1": 210.0,
```

```

        "y1": 530.0,
        "x2": 710.0,
        "y2": 425.0,
        "name": "wire20"
    },
    {
        "type": "elbow",
        "wire_dir": "H",
        "node_num": "0",
        "x1": 210.0,
        "y1": 530.0,
        "x2": 550.0,
        "y2": 485.0,
        "name": "wire21"
    },
    {
        "type": "elbow",
        "wire_dir": "H",
        "node_num": "0",
        "x1": 210.0,
        "y1": 530.0,
        "x2": 400.0,
        "y2": 440.0,
        "name": "wire22"
    }
]
}
]

```

Final source code

Comp_Lib/ __init__.py

```

# Import lumped components
from Comp_Lib.analog_component import AnalogComponent

# Import connector related classes
from Comp_Lib.connector import Connector
from Comp_Lib.connection import Connection

```

Comp_Lib/component.py

```
import tkinter as tk
from PIL import Image, ImageTk
from pathlib import Path

from Helper_Lib import Point


class Component:
    def __init__(self, canvas, comp_type, x1, y1, value):
        """Base class for component classes"""
        self.canvas = canvas
        self.comp_type = comp_type
        self.x1 = x1
        self.y1 = y1
        self.value = value
        self.units = None

        if self.comp_type == 'npn_transistor':
            self.bf = None
            self.cjc = None
            self.cjc_units = None
            self.rb = None

        self.id = None
        self.sel_id = None

        self.is_selected = False
        self.is_drawing = False
        self.selector = None
        self.angle = 0
        self.text = None
        self.comp_text = None
        self.text_id = None

        self.filename = None
        self.a_image = None
        self.ph_image = None
        self.bbox = None

        self.out = None
        self.in1 = None
        self.ba = None
        self.em = None
```

```

    self.co = None

    self.conn_list = []
    self.wire_list = []

    # Define component parameters in a dictionary of dictionaries
    self.params = {
        'filename': {
            'resistor': "../images/lumped/resistor_60x30.png",
            'capacitor': "../images/lumped/capacitor_60x30.png",
            'inductor': "../images/lumped/inductor_60x30.png",
            'ground': "../images/sources/ground_50x40.png",
            'dc_source': "../images/sources/voltage_source_40x40.png",
            'isource': "../images/sources/current_source_40x40.png",
            'ac_source': "../images/sources/ac_voltage_source_40x40.png",
            'npn_transistor':
                ".../images/transistors/npn_transistor_60x71.png"
        },
        'text': {
            'resistor': 'R1',
            'capacitor': 'C1',
            'inductor': 'L1',
            'ground': "",
            'dc_source': "Vpwr",
            'isource': "I",
            'ac_source': "Vin",
            'npn_transistor': "Q1"
        }
    }

    def set_image_filename(self):
        self.filename = Path(__file__).parent / self.params['filename'][self.comp_type]

    def create_text(self):
        self.comp_text = self.params['text'][self.comp_type]
        if self.comp_type == 'isource' or self.comp_type == 'dc_source' or
        self.comp_type == 'ac_source':
            text_loc = Point(self.x1-40, self.y1) # Put text on left side of
symbol
        elif self.comp_type == 'npn_transistor':
            text_loc = Point(self.x1 - 10, self.y1 - 40) # Put text above
symbol
        else:
            text_loc = Point(self.x1, self.y1-30) # Put text above symbol
        self.text_id = self.canvas.create_text(text_loc.x, text_loc.y,

```

```

        text=self.comp_text, fill="black",
        font='Helvetica 10 bold',
        angle=self.angle, tags="text")

def create_image(self, filename):
    """Initial component image creation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True)
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags='gate')

def update_position(self):
    """Update the position when the gate object is moved"""
    self.canvas.coords(self.id, self.x1, self.y1) # Update position

def update_image(self, filename):
    """Update the image for gate symbol rotation"""
    self.a_image = Image.open(filename)
    self.a_image = self.a_image.rotate(self.angle, expand=True) # Update
image rotation
    self.ph_image = ImageTk.PhotoImage(self.a_image)
    self.canvas.itemconfig(self.id, image=self.ph_image) # Update image

def update_bbox(self):
    """Update the bounding box to get current gate coordinates"""
    self.bbox = self.canvas.bbox(self.id)

def update_text(self):
    self.canvas.itemconfig(self.text_id, text=self.comp_text)
    if self.comp_type == 'isource' or self.comp_type == 'dc_source' or
self.comp_type == 'ac_source':
        if self.angle == 0 or self.angle == 180:
            self.set_east_text()
        elif self.angle == 90 or self.angle == 270:
            self.set_north_text()
    elif self.comp_type == 'npn_transistor':
        self.set_transistor_text()
    else:
        if self.angle == 0 or self.angle == 180:
            self.set_north_text()
        elif self.angle == 90 or self.angle == 270:
            self.set_east_text()

def set_east_text(self):
    self.canvas.coords(self.text_id, self.x1 - 70, self.y1)

```

```

def set_north_text(self):
    self.canvas.coords(self.text_id, self.x1, self.y1 - 30)

def set_transistor_text(self):
    self.canvas.itemconfig(self.text_id, text=self.text)
    self.canvas.coords(self.text_id, self.x1 - 10, self.y1 - 40)

def create_selector(self):
    """Create the red rectangle selector and check to see if the gate is
selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
    self.bbox[3] + 5
    self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
    self.set_selector_visibility()

def update_selector(self):
    """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
    self.bbox[3] + 5
    self.canvas.coords(self.sel_id, x1, y1, x2, y2)
    self.set_selector_visibility()

def set_selector_visibility(self):
    """Set the selector visibility state"""
    if self.is_selected:
        self.canvas.itemconfig(self.sel_id, state='normal')
    else:
        self.canvas.itemconfig(self.sel_id, state='hidden')

def set_connector_visibility(self):
    """Set the connector visibility state"""
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

def check_connector_hit(self, x, y):
    """Hit test to see if a connector is at the provided x, y
coordinates"""
    for conn in self.conn_list:
        if conn.connector_hit_test(x, y):

```

```

        return conn
    return None

def move_connected_wires(self):
    """Resize connected wires if the shape is moved"""
    for connection in self.wire_list: # comp_conn, wire_name, wire_end
        for connector in self.conn_list:
            if connector.name == connection.comp_conn:
                wire_obj = self.canvas.wire_dict[connection.wire_name]
                if connection.wire_end == "begin":
                    wire_obj.x1 = connector.x
                    wire_obj.y1 = connector.y
                elif connection.wire_end == "end":
                    wire_obj.x2 = connector.x
                    wire_obj.y2 = connector.y

def rotate(self):
    """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
if angle > 270 deg"""
    self.angle += 90
    if self.angle > 270:
        self.angle = 0

def hit_test(self, x, y):
    x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
    if x1 <= x <= x2 and y1 <= y <= y2:
        self.is_selected = True
    else:
        self.is_selected = False

```

Comp_Lib/analog_component.py

```

from Comp_Lib.component import Component
from Comp_Lib.connector import Connector
from Helper_Lib import Point

class AnalogComponent(Component):
    """Universal model for analog components"""
    def __init__(self, canvas, comp_type, x1, y1, value=0):
        super().__init__(canvas, comp_type, x1, y1, value)

        self.conn_params = {

```

```

'ports': {
    'resistor': 2,
    'capacitor': 2,
    'inductor': 2,
    'ground': 1,
    'dc_source': 2,
    'isource': 2,
    'ac_source': 2,
    'npn_transistor': 3
},
'conn_loc': {
    'resistor': 'ew',
    'capacitor': 'ew',
    'inductor': 'ew',
    'ground': 'n',
    'dc_source': 'ns',
    'isource': 'ns',
    'ac_source': 'ns',
    'npn_transistor': 'nsw'
}
}

self.create()

def create(self):
    self.set_image_filename()
    self.create_image(self.filename)
    self.update_bbox()
    self.create_text()
    self.create_selector()
    self.create_connectors()
    self.set_connector_visibility()

def update(self):
    self.update_position()
    self.update_image(self.filename)
    self.update_bbox()
    self.update_text()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def create_connectors(self):
    # Calculate position of connectors from current comp position and size
    center, e, w, n, s = self.get_geometry()

```

```

num_ports = self.conn_params['ports'][self.comp_type]
if num_ports == 1:
    self.out = Connector(self.canvas, "in1", center.x, center.y)
    self.conn_list = [self.out]
elif num_ports == 2:
    self.out = Connector(self.canvas, "out", center.x, center.y)
    self.in1 = Connector(self.canvas, "in1", center.x, center.y)
    self.conn_list = [self.in1, self.out]
elif num_ports == 3:
    self.ba = Connector(self.canvas, "base", center.x, center.y)
    self.em = Connector(self.canvas, "emitter", center.x, center.y)
    self.co = Connector(self.canvas, "collector", center.x, center.y)
    self.conn_list = [self.ba, self.em, self.co]

def update_connectors(self):
    """Update the position of connectors here"""
    center, e, w, n, s = self.get_geometry()

    conn_loc = self.conn_params['conn_loc'][self.comp_type]
    if conn_loc == 'ew': # 2-port with ew ports
        self.calc_ew_conn_rotation(n, s, e, w)
    elif conn_loc == 'ns': # 2-port with ns ports
        self.calc_ns_conn_rotation(n, s, e, w)
    elif conn_loc == 'n': # 1-port with n port
        self.calc_n_conn_rotation(n, w)
    elif conn_loc == 'nsw': # 3-port with nsw ports
        self.calc_nsw_conn_rotation(n, s, e, w)

    for c in self.conn_list:
        c.update()

    self.move_connected_wires()

def calc_ew_conn_rotation(self, n, s, e, w):
    if self.angle == 0 or self.angle == 180:
        self.out.x, self.out.y = w.x, w.y
        self.in1.x, self.in1.y = e.x, e.y
    elif self.angle == 90 or self.angle == 270:
        self.out.x, self.out.y = n.x, n.y
        self.in1.x, self.in1.y = s.x, s.y

def calc_ns_conn_rotation(self, n, s, e, w):
    if self.angle == 0 or self.angle == 180:
        self.out.x, self.out.y = n.x, n.y
        self.in1.x, self.in1.y = s.x, s.y
    elif self.angle == 90 or self.angle == 270:

```

```

        self.out.x, self.out.y = w.x, w.y
        self.in1.x, self.in1.y = e.x, e.y

    def calc_n_conn_rotation(self, n, w):
        if self.angle == 0 or self.angle == 180:
            self.out.x, self.out.y = n.x, n.y
        elif self.angle == 90 or self.angle == 270:
            self.out.x, self.out.y = w.x, w.y

    def calc_nsw_conn_rotation(self, n, s, e, w):
        if self.angle == 0 or self.angle == 180:
            self.em.x, self.em.y = n.x, n.y
            self.co.x, self.co.y = s.x, s.y
            self.ba.x, self.ba.y = w.x, w.y
        elif self.angle == 90 or self.angle == 270:
            self.em.x, self.em.y = w.x, w.y
            self.co.x, self.co.y = e.x, e.y
            self.ba.x, self.ba.y = s.x, s.y

    def get_geometry(self):
        sign = lambda angle: 1 if angle == 0 or angle == 180 else -1

        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

        if self.comp_type == 'npn_transistor':
            e = Point(center.x + sign(self.angle) * w / 2, center.y)
            w = Point(center.x - sign(self.angle) * w / 2, center.y)
            n = Point(center.x + 20, center.y - sign(self.angle) * h / 2)
            s = Point(center.x + 20, center.y + sign(self.angle) * h / 2)
        else:
            e = Point(center.x - sign(self.angle) * w / 2, center.y)
            w = Point(center.x + sign(self.angle) * w / 2, center.y)
            n = Point(center.x, center.y - sign(self.angle) * h / 2)
            s = Point(center.x, center.y + sign(self.angle) * h / 2)

        return center, e, w, n, s

    def __repr__(self):
        if self.comp_type == "npn_transistor":
            return ("Type: " + self.comp_type + " Comp Text: " + self.comp_text
+ " x1: " + str(self.x1) +
                    " y1: " + str(self.y1) + " bf: " + str(self.bf) + " cjc: "
+ str(self.cjc) + " cjc_units: " +

```

```

        self.cjc_units + " rb: " + str(self.rb) + " wire list: " +
str(self.wire_list.__repr__()))
    else:
        return ("Type: " + self.comp_type + " Comp Text: " + self.comp_text
+ " x1: " + str(self.x1) + " y1: "
            + str(self.y1) + " value: " + str(self.value) + " wire
list: " + str(self.wire_list.__repr__()))

def reprJson(self):
    if self.comp_type == "npn_transistor":
        return dict(type=self.comp_type, text=self.text, x1=self.x1,
y1=self.y1, angle=self.angle,
                    bf=self.bf, cjc=self.cjc, cjc_units=self.cjc_units,
rb=self.rb, wire_list=self.wire_list)
    else:
        return dict(type=self.comp_type, text=self.text, x1=self.x1,
y1=self.y1, angle=self.angle,
                    value=self.value, units=self.units,
wire_list=self.wire_list)

```

Comp_Lib/connection.py

```

class Connection:
    def __init__(self, comp_conn, wire_name, wire_end):
        self.type = "connection"
        self.comp_conn = comp_conn      # in1, out
        self.wire_name = wire_name      # wire string name
        self.wire_end = wire_end        # "begin" or "end"

    def __repr__(self):
        return "Connector Name: " + self.comp_conn + \
               " Wire Name: " + self.wire_name + \
               " Wire End: " + self.wire_end

    def reprJson(self):
        return dict(type=self.type, comp_conn=self.comp_conn,
wire_name=self.wire_name, wire_end=self.wire_end)

```

Comp_Lib/connector.py

```

class Connector:
    def __init__(self, canvas, name, x, y):
        """Connector class"""
        self.canvas = canvas
        self.name = name
        self.x = x
        self.y = y

        self.id = None

        self.radius = 5
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
                                             self.radius,
                                             self.x + self.radius, self.y +
                                             self.radius)

        self.create_connector()

    def create_connector(self):
        # Create the connector here
        points = [self.x - self.radius, self.y - self.radius, self.x +
                  self.radius, self.y + self.radius]
        self.id = self.canvas.create_oval(points, fill="white",
                                         outline="black", width=2, tags='connector')

    def update(self):
        """Update the connector here"""
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
                                              self.radius,
                                              self.x + self.radius, self.y +
                                              self.radius)
        points = [self.x - self.radius, self.y - self.radius, self.x +
                  self.radius, self.y + self.radius]
        self.canvas.coords(self.id, points)

    def set_pos(self, x, y):
        """Set the connector position here"""
        self.x = x
        self.y = y

    def connector_hit_test(self, x, y):
        """Connector hit test"""
        if self.x1 <= x <= self.x2 and self.y1 <= y <= self.y2:
            return True
        else:
            return False

```

```
def __repr__(self):
    return ("Connector: " + self.name + " (" + str(self.x1) + ", " +
str(self.y1) + ")" +
        " (" + str(self.x2) + ", " + str(self.y2) + ")")
```

Helper_Lib/__init__.py

```
# Import helper classes
from .point import Point
```

Helper_Lib/point.py

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def set(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"

if __name__ == "__main__":
    # Test block
    pt1 = Point(10, 10)
    pt2 = Point(100, 100)
    print("pt1: ", pt1)
    print("pt2: ", pt2)
```

UI_Lib/__init__.py

```
# Import custom frame and canvas classes
from UI_Lib.left_frame import LeftFrame
```

```

from UI_Lib.top_frame import TopFrame
from UI_Lib.canvas import Canvas

# Import custom button frame classes
from UI_Lib.lump_button_frame import LumpButtonFrame
from UI_Lib.sources_button_frame import SourcesButtonFrame
from UI_Lib.wire_button_frame import WireButtonFrame

# Import custom classes
from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid

```

UI_Lib/active_button_frame.py

```

import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from Comp_Lib import AnalogComponent


class ActiveButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None

        self.button_list = [("npn_transistor", "../icons/npn-transistor.png")]

        self.init_frame_widgets()

    def init_frame_widgets(self):
        frame_name_label = ctk.CTkLabel(self, text="Active", font=("Helvetica", 10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W, padx=2, pady=2)

        row_num, col_num = 1, 0
        for button in self.button_list:
            a_image = ctk.CTkImage(light_image=Image.open
                                  (Path(__file__).parent / button[1])),

```

```

        dark_image=Image.open
        (Path(__file__).parent / button[1]),
        size=(24, 24))
    self.button_id = ctk.CTkButton(self, text="", image=a_image,
width=30,
                           command=lambda
a_name=button[0]:self.create_events(a_name))
    self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W,
padx=2, pady=2)
    ToolTip(self.button_id, msg=button[0])
    row_num, col_num = self.update_grid_numbers(row_num, col_num)

def create_events(self, name):
    comp = None
    if name == "npn_transistor":
        comp = AnalogComponent(self.canvas, 'npn_transistor', 150, 150)
    self.canvas.comp_list.append(comp)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

@staticmethod
def update_grid_numbers(row, column):
    column += 1
    if column > 2:
        column = 0
        row += 1
    return row, column

```

UI_Lib/canvas.py

```

import customtkinter as ctk

from UI_Lib.mouse import Mouse
from UI_Lib.grid import Grid
from Wire_Lib import Node


class CompDialog(ctk.CTkToplevel):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.geometry("400x300")
        self.title("Component Dialog")

```

```
    self.text = ""
    self.value = 0
    self.units = ""

    self.text_entry = None
    self.value_entry = None
    self.units_entry = None

    self.initDialog()

def initDialog(self):
    text_frame = ctk.CTkFrame(self)
    text_frame.pack(fill=ctk.X)

    text_lbl = ctk.CTkLabel(text_frame, text="Text", width=6)
    text_lbl.pack(side=ctk.LEFT, padx=5, pady=10)

    self.text_entry = ctk.CTkEntry(text_frame, placeholder_text="CTkEntry")
    self.text_entry.pack(fill=ctk.X, padx=5, expand=True)

    value_frame = ctk.CTkFrame(self)
    value_frame.pack(fill=ctk.X)

    value_lbl = ctk.CTkLabel(value_frame, text="Value", width=6)
    value_lbl.pack(side=ctk.LEFT, padx=5, pady=10)

    self.value_entry = ctk.CTkEntry(value_frame,
placeholder_text="CTkEntry")
    self.value_entry.pack(fill=ctk.X, padx=5, expand=True)

    units_frame = ctk.CTkFrame(self)
    units_frame.pack(fill=ctk.X)

    units_lbl = ctk.CTkLabel(units_frame, text="Units (\u03a9,nF,pF, etc.)",
width=6)
    units_lbl.pack(side=ctk.LEFT, padx=5, pady=10)

    self.units_entry = ctk.CTkEntry(units_frame,
placeholder_text="CTkEntry")
    self.units_entry.pack(fill=ctk.X, padx=5, expand=True)

    frame9 = ctk.CTkFrame(self)
    frame9.pack(fill=ctk.X)

    btn = ctk.CTkButton(frame9, text="Cancel", command=self.cancel,
width=60)
```

```
btn.pack(side=ctk.RIGHT, padx=5, pady=5)

btn = ctk.CTkButton(frame9, text="OK", command=self.onSubmit, width=60)
btn.pack(side=ctk.RIGHT, padx=5, pady=5)

def onSubmit(self):
    self.text = self.text_entry.get()
    self.value = self.value_entry.get()
    self.units = self.units_entry.get()
    self.destroy()

def cancel(self):
    self.destroy()

class TransistorDialog(ctk.CTkToplevel):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.geometry("400x300")
        self.title("Transistor Dialog")

        self.text = ""
        self.output1 = ""
        self.output2 = ""
        self.output3 = ""
        self.output4 = ""

        self.text_entry = None
        self.entry1 = None
        self.entry2 = None
        self.entry3 = None
        self.entry4 = None

        self.initDialog()

    def initDialog(self):
        text_frame = ctk.CTkFrame(self)
        text_frame.pack(fill=ctk.X)

        text_lbl = ctk.CTkLabel(text_frame, text="Text", width=6)
        text_lbl.pack(side=ctk.LEFT, padx=5, pady=10)

        self.text_entry = ctk.CTkEntry(text_frame, placeholder_text="CTkEntry")
        self.text_entry.pack(fill=ctk.X, padx=5, expand=True)

        frame1 = ctk.CTkFrame(self)
```

```
frame1.pack(fill=ctk.X)

lbl1 = ctk.CTkLabel(frame1, text="bf", width=6)
lbl1.pack(side=ctk.LEFT, padx=5, pady=10)

self.entry1 = ctk.CTkEntry(frame1, placeholder_text="CTkEntry")
self.entry1.pack(fill=ctk.X, padx=5, expand=True)

frame2 = ctk.CTkFrame(self)
frame2.pack(fill=ctk.X)

lbl2 = ctk.CTkLabel(frame2, text="cjc", width=6)
lbl2.pack(side=ctk.LEFT, padx=5, pady=10)

self.entry2 = ctk.CTkEntry(frame2, placeholder_text="CTkEntry")
self.entry2.pack(fill=ctk.X, padx=5, expand=True)

frame3 = ctk.CTkFrame(self)
frame3.pack(fill=ctk.X)

lbl3 = ctk.CTkLabel(frame3, text="cjc_units", width=6)
lbl3.pack(side=ctk.LEFT, padx=5, pady=10)

self.entry3 = ctk.CTkEntry(frame3, placeholder_text="CTkEntry")
self.entry3.pack(fill=ctk.X, padx=5, expand=True)

frame4 = ctk.CTkFrame(self)
frame4.pack(fill=ctk.X)

lbl4 = ctk.CTkLabel(frame4, text="rb", width=6)
lbl4.pack(side=ctk.LEFT, padx=5, pady=10)

self.entry4 = ctk.CTkEntry(frame4, placeholder_text="CTkEntry")
self.entry4.pack(fill=ctk.X, padx=5, expand=True)

frame9 = ctk.CTkFrame(self)
frame9.pack(fill=ctk.X)

btn = ctk.CTkButton(frame9, text="Cancel", command=self.cancel,
width=60)
btn.pack(side=ctk.RIGHT, padx=5, pady=5)

# Command tells the form what to do when the button is clicked
btn = ctk.CTkButton(frame9, text="OK", command=self.onSubmit, width=60)
btn.pack(side=ctk.RIGHT, padx=5, pady=5)
```

```
def onSubmit(self):
    self.text = self.text_entry.get()
    self.output1 = self.entry1.get()
    self.output2 = self.entry2.get()
    self.output3 = self.entry3.get()
    self.output4 = self.entry4.get()
    self.destroy()

def cancel(self):
    self.destroy()

class Canvas(ctk.CTkCanvas):
    def __init__(self, parent):
        super().__init__(parent)
        self.led_color = "red"
        self.led_size = "large"
        self.grid = Grid(self, 10)
        self.wire_dir = "H"
        self.mouse = Mouse(self)
        self.comp_list = []
        self.wire_list = []
        self.wire_dict = {}

        self.mouse.move_mouse_bind_events()

    def redraw(self):
        self.delete('grid_line')
        self.grid.draw()
        self.tag_lower("grid_line")
        for c in self.comp_list:
            c.update()

    def show_connectors(self):
        for s in self.comp_list:
            s.is_drawing = True
        self.redraw()

    def hide_connectors(self):
        for s in self.comp_list:
            s.is_drawing = False
        self.redraw()

    def edit_shape(self, _event=None):
        if self.mouse.selected_comp:
            comp = self.mouse.selected_comp
```

```

        if isinstance(comp, Node):
            dialog = ctk.CTkInputDialog(text="Enter new text", title="Edit
Node Text")
            comp.text = dialog.get_input()
            self.redraw()
        elif (comp.comp_type == 'resistor' or comp.comp_type == 'inductor'
or comp.comp_type == 'capacitor' or
            comp.comp_type == 'dc_source' or comp.comp_type ==
'ac_source' or
            comp.comp_type == 'isource'):
            dialog = CompDialog(self)
            dialog.wm_attributes('-topmost', True)
            dialog.wait_window()
            comp.text = dialog.text
            comp.value = dialog.value
            comp.units = dialog.units
            comp.comp_text = comp.text + "=" + str(comp.value) + comp.units
            self.redraw()
        elif comp.comp_type == 'npn_transistor':
            dialog = TransistorDialog(self)
            dialog.wm_attributes('-topmost', True)
            dialog.wait_window()
            comp.text = dialog.text
            comp.bf = dialog.output1
            comp.cjc = dialog.output2
            comp.cjc_units = dialog.output3
            comp.rb = dialog.output4
            self.redraw()

    def set_horiz_dir(self, _event):
        self.wire_dir = "H"

    def set_vert_dir(self, _event):
        self.wire_dir = "V"

```

UI_Lib/file_menu_frame.py

```

import customtkinter as ctk
from tkinter import filedialog as fd
from pathlib import Path
import json
from PIL import Image

```

```

from Comp_Lib import Connection, AnalogComponent
from Wire_Lib import AnalogWire, Node


class Encoder(json.JSONEncoder):
    def default(self, o):
        if hasattr(o, "reprJson"):
            return o.reprJson()
        else:
            return super().default(o)


class Decoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self, object_hook=Decoder.from_dict)

    @staticmethod
    def from_dict(_d):
        return _d


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.obj_type_dict = {'resistor': AnalogComponent,
                             'inductor': AnalogComponent,
                             'capacitor': AnalogComponent,
                             'ground': AnalogComponent,
                             'dc_source': AnalogComponent,
                             'isource': AnalogComponent,
                             'ac_source': AnalogComponent,
                             'npn_transistor': AnalogComponent,
                             'straight': AnalogWire,
                             'segment': AnalogWire,
                             'elbow': AnalogWire}

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
                               command=self.new_diagram)
        new_btn.pack(pady=5)

```

```

        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
command=window.destroy)
        exit_btn.pack(pady=5)

        my_image = ctk.CTkImage(light_image=Image.open
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                                dark_image=Image.open
                                (Path(__file__).parent /
"../icons/hamburger_menu.png"),
                                size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def new_diagram(self):
        self.canvas.delete("all")
        self.canvas.comp_list.clear()
        self.canvas.wire_list.clear()

    def load_diagram(self):
        try:
            filetypes = (('json files', '*.json'), ('All files', '*.*'))
            f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
            with open(f) as file:
                d = json.load(file)
                self.convert_json_data(d)
        except FileNotFoundError:
            with open('untitled.canvas', 'w') as _file:
                pass

    def convert_json_data(self, data):
        """Convert json data to a circuit object"""
        # Get circuit list from json data
        json_comp_list = data[0]['comp_list']
        for json_comp in json_comp_list:
            if (json_comp['type'] == 'resistor' or json_comp['type'] ==

```

```

'inductor' or json_comp['type'] == 'capacitor'
        or json_comp['type'] == 'dc_source' or json_comp['type'] ==
'isource'
            or json_comp['type'] == 'ac_source' or json_comp['type'] ==
'ground'):
    comp = AnalogComponent(self.canvas, json_comp['type'],
int(json_comp['x1']), int(json_comp['y1']),
float(json_comp['value']))
    comp.text = json_comp['text']
    comp.units = json_comp['units']
    if comp.value:
        comp.comp_text = comp.text + "=" + str(comp.value) +
comp.units
        comp.angle = json_comp['angle']
        conn_dict = json_comp['wire_list'][0]
        comp.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
conn_dict['wire_end']))
    if not json_comp['type'] == 'ground':
        conn_dict = json_comp['wire_list'][1]
        comp.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
conn_dict['wire_end']))
    self.canvas.comp_list.append(comp)
elif json_comp['type'] == 'npn_transistor':
    comp = AnalogComponent(self.canvas, json_comp['type'],
int(json_comp['x1']), int(json_comp['y1']), 0)
    comp.text = json_comp['text']
    comp.bf = json_comp['bf']
    comp.cjc = json_comp['cjc']
    comp.cjc_units = json_comp['cjc_units']
    comp.rb = json_comp['rb']
    comp.angle = json_comp['angle']
    conn_dict = json_comp['wire_list'][0]
    comp.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
conn_dict['wire_end']))
    conn_dict = json_comp['wire_list'][1]
    comp.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
conn_dict['wire_end']))
    conn_dict = json_comp['wire_list'][2]
    comp.wire_list.append(Connection(conn_dict['comp_conn'],
conn_dict['wire_name'],
conn_dict['wire_end']))
    self.canvas.comp_list.append(comp)

```

```

        elif json_comp['type'] == 'straight' or json_comp['type'] ==
'segment' or json_comp['type'] == 'elbow':
            self.canvas.wire_dir = json_comp['wire_dir']
            wire = AnalogWire(self.canvas, json_comp['type'],
int(json_comp['x1']), int(json_comp['y1']),
                           int(json_comp['x2']), int(json_comp['y2']))
            wire.name = json_comp['name']
            wire.node_num = json_comp['node_num']
            self.canvas.comp_list.append(wire)
            self.canvas.wire_dict[wire.name] = wire
        elif json_comp['type'] == 'node':
            node = Node(self.canvas, int(json_comp['x1']),
int(json_comp['y1']))
            node.text = json_comp['text']
            for connection in json_comp['wire_list']:
                node.wire_list.append(Connection(connection['comp_conn'],
connection['wire_name'],
                           connection['wire_end']))
            self.canvas.comp_list.append(node)

# for comp in self.canvas.comp_list:
#     print(comp)
self.canvas.redraw()

def save_diagram(self):
    comp_dict = {'comp_list': self.canvas.comp_list}
    circuit = [comp_dict]

    filetypes = (('json files', '*.json'), ('All files', '*.*'))
    f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
    with open(f, 'w') as file:
        file.write(json.dumps(circuit, cls=Encoder, indent=4))

def show_menu(self):
    if not self.menu_on:
        self.menu_frame.place(x=5, y=60)
        self.menu_frame.tkraise()
        self.menu_on = True
    else:
        self.menu_frame.place_forget()
        self.menu_on = False

```

```

class Grid:
    def __init__(self, canvas, grid_size):
        self.canvas = canvas
        self.grid_size = grid_size
        self.grid_visible = True

        self.grid_snap = 5
        self.draw()

    def draw(self):
        if self.grid_visible:
            w = self.canvas.winfo_width() # Get current width of canvas
            h = self.canvas.winfo_height() # Get current height of canvas

            # Creates all vertical lines at intervals of 100
            for i in range(0, w, self.grid_size):
                self.canvas.create_line([(i, 0), (i, h)], fill="#cccccc",
tags='grid_line')

            # Creates all horizontal lines at intervals of 100
            for i in range(0, h, self.grid_size):
                self.canvas.create_line([(0, i), (w, i)], fill="#cccccc",
tags='grid_line')

    def snap_to_grid(self, x, y):
        if self.grid_visible:
            x = round(x / self.grid_snap) * self.grid_snap
            y = round(y / self.grid_snap) * self.grid_snap
        return x, y

```

UI_Lib/help_frame.py

```

import customtkinter as ctk
from tkinter import messagebox
from PIL import Image

class HelpFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.window = window
        self.parent = parent
        self.canvas = canvas

```

```
    self.menu_on = False

    self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

    about_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/about.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/about.png"),
size=(24, 24))

    about_button = ctk.CTkButton(self.menu_frame, text="About",
image=about_image, width=30,
command=self.show_about_dialog)
    about_button.pack(side=ctk.TOP, padx=5, pady=5)

    my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/help.png"),
dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/help.png"),
size=(24, 24))

    button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
    button.pack(side=ctk.LEFT, padx=5, pady=10)

def show_menu(self):
    if not self.menu_on:
        menu_pos_x = self.canvas.winfo_width()
        self.menu_frame.place(x=menu_pos_x + 50, y=60)
        self.menu_frame.tkraise()
        self.menu_on = True
    else:
        self.menu_frame.place_forget()
        self.menu_on = False

@staticmethod
def show_about_dialog():
    messagebox.showinfo("About Digital Simulator", "RF/Microwave Simulator
v0.1\n" +
"Author: Rick A. Crist\n" + "2023")
```

UI.Lib/left_frame.py

```
import customtkinter as ctk
from UI.Lib.lump_button_frame import LumpButtonFrame
from UI.Lib.sources_button_frame import SourcesButtonFrame
from UI.Lib.active_button_frame import ActiveButtonFrame
from UI.Lib.wire_button_frame import WireButtonFrame


class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas

        self.comp_button_frame = LumpButtonFrame(self, self.canvas)
        self.comp_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.sources_button_frame = SourcesButtonFrame(self, self.canvas)
        self.sources_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.active_button_frame = ActiveButtonFrame(self, self.canvas)
        self.active_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        self.wire_button_frame = WireButtonFrame(self, self.canvas)
        self.wire_button_frame.pack(side=ctk.TOP, padx=5, pady=5)
```

UI.Lib/lump_button_frame.py

```
import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

from Comp.Lib import AnalogComponent


class LumpButtonFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas
        self.button_id = None
```

```

        self.button_list = [("resistor", "../icons/resistor.png"),
                            ("inductor", "../icons/inductor.png"),
                            ("capacitor", "../icons/capacitor.png")]

    self.init_frame_widgets()

    def init_frame_widgets(self):
        frame_name_label = ctk.CTkLabel(self, text="Lumped", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        row_num, col_num = 1, 0
        for button in self.button_list:
            a_image = ctk.CTkImage(light_image=Image.open
                                    (Path(__file__).parent / button[1]),
                                    dark_image=Image.open
                                    (Path(__file__).parent / button[1]),
                                    size=(24, 24))
            self.button_id = ctk.CTkButton(self, text="", image=a_image,
width=30,
                                         command=lambda
a_name=button[0]:self.create_events(a_name))
            self.button_id.grid(row=row_num, column=col_num, sticky=ctk.W,
padx=2, pady=2)
            ToolTip(self.button_id, msg=button[0])
            row_num, col_num = self.update_grid_numbers(row_num, col_num)

    def create_events(self, name):
        comp = None
        if name == "resistor":
            comp = AnalogComponent(self.canvas, 'resistor', 100, 100, 75)
        elif name == "inductor":
            comp = AnalogComponent(self.canvas, 'inductor', 100, 100, 5)
        elif name == "capacitor":
            comp = AnalogComponent(self.canvas, 'capacitor', 100, 100, 1)
        self.canvas.comp_list.append(comp)
        self.canvas.redraw()
        self.canvas.mouse.move_mouse_bind_events()

    @staticmethod
    def update_grid_numbers(row, column):
        column += 1
        if column > 2:
            column = 0

```

```
    row += 1
    return row, column
```

UI_Lib/mouse.py

```
from Helper_Lib import Point
from Wire_Lib import AnalogWire, Node
from Comp_Lib import Connection

class Mouse:
    def __init__(self, canvas):
        self.canvas = canvas

        self.selected_comp = None
        self.current_wire_obj = None
        self.start = Point(0, 0)
        self.offset1 = Point(0, 0)
        self.offset2 = Point(0, 0)

        self.move_mouse_bind_events()

    def move_mouse_bind_events(self):
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def draw_wire_mouse_events(self): # Added method to bind draw wire methods
        self.canvas.bind("<Button-1>", self.draw_left_down)
        self.canvas.bind("<B1-Motion>", self.draw_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)

    def resize_wire_mouse_events(self):
        self.canvas.bind("<Button-1>", self.resize_left_down)
        self.canvas.bind("<B1-Motion>", self.resize_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.comp_hit_test(x, y)
        if self.selected_comp:
            if isinstance(self.selected_comp, AnalogWire):
                x1, y1 = self.selected_comp.x1, self.selected_comp.y1
```

```

        x2, y2 = self.selected_comp.x2, self.selected_comp.y2
        self.offset1.set(x - x1, y - y1)
        self.offset2.set(x - x2, y - y2)
    else:
        x1, y1 = self.selected_comp.x1, self.selected_comp.y1
        self.offset1.x, self.offset1.y =
self.canvas.grid.snap_to_grid(self.offset1.x, self.offset1.y)
        self.offset1.set(x - x1, y - y1)

def move_left_drag(self, event):
    if self.selected_comp:
        if isinstance(self.selected_comp, AnalogWire):
            x1 = event.x - self.offset1.x
            y1 = event.y - self.offset1.y
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
            x2 = event.x - self.offset2.x
            y2 = event.y - self.offset2.y
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
            self.selected_comp.x1, self.selected_comp.y1 = x1, y1
            self.selected_comp.x2, self.selected_comp.y2 = x2, y2
            self.canvas.redraw()
        else:
            x = event.x - self.offset1.x
            y = event.y - self.offset1.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            self.selected_comp.x1, self.selected_comp.y1 = x, y
            self.canvas.redraw()

def move_left_up(self, _event):
    self.offset1.set(0, 0)
    self.offset2.set(0, 0)
    self.canvas.redraw()

def draw_left_down(self, event): # Added method for draw left down
    if self.current_wire_obj:
        # self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y
        self.start.x, self.start.y =
self.canvas.grid.snap_to_grid(self.start.x, self.start.y)

        self.current_wire_obj.x1, self.current_wire_obj.y1 = self.start.x,
self.start.y
        self.current_wire_obj.x2, self.current_wire_obj.y2 = self.start.x,
self.start.y

```

```

        self.select_connector(self.current_wire_obj, "begin", self.start.x,
self.start.y)

    def draw_left_drag(self, event): # Added method for draw left drag
        if self.current_wire_obj:
            wire = self.current_wire_obj
            x, y = event.x, event.y
            x, y = self.canvas.grid.snap_to_grid(x, y)
            wire.x1, wire.y1 = self.start.x, self.start.y
            wire.x2, wire.y2 = x, y
            self.current_wire_obj.update()

    def draw_left_up(self, event): # Added method for draw left up
        self.select_connector(self.current_wire_obj, "end", event.x, event.y)
        self.canvas.hide_connectors()
        self.current_wire_obj = None
        self.move_mouse_bind_events()

    def resize_left_down(self, event):
        if self.selected_comp:
            x1, y1 = self.selected_comp.x1, self.selected_comp.y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1, "resize")
            x2, y2 = self.selected_comp.x2, self.selected_comp.y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2, "resize")
            self.offset1.x = event.x - x1
            self.offset1.y = event.y - y1
            self.offset2.x = event.x - x2
            self.offset2.y = event.y - y2
            self.selected_comp.update()

    def resize_left_drag(self, event):
        if self.selected_comp:
            offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
self.offset2.y]
            self.selected_comp.resize(offsets, event)
            self.selected_comp.update()

    def resize_left_up(self, _event):
        self.offset1.x, self.offset1.y = 0, 0
        self.offset2.x, self.offset2.y = 0, 0
        self.move_mouse_bind_events()

    def comp_hit_test(self, x, y):
        for comp in self.canvas.comp_list:
            comp.hit_test(x, y)
            if comp.is_selected:

```

```

        if isinstance(self.selected_comp, AnalogWire):
            result = comp.sel_hit_test(x, y)
            if result is not None:
                self.resize_wire_mouse_events()
            comp.update()
            self.selected_comp = comp
            return

    # No shape hit - unselect all
    self.selected_comp = None
    self.unselect_all()

def unselect_all(self):
    for comp in self.canvas.comp_list:
        comp.is_selected = False
        comp.update()

def select_connector(self, wire_obj, wire_end, x, y):
    for comp in self.canvas.comp_list:
        if not isinstance(comp, AnalogWire):
            conn = comp.check_connector_hit(x, y)
            if conn:
                if wire_end == "begin":
                    wire_obj.x1, wire_obj.y1 = conn.x, conn.y
                elif wire_end == "end":
                    wire_obj.x2, wire_obj.y2 = conn.x, conn.y
                a_conn = Connection(conn.name, self.current_wire_obj.name,
wire_end)
                # wire_obj.create_wire_list_cnx(comp.comp_type, conn.name)
                if isinstance(comp, Node):
                    wire_obj.set_node_num(comp.text)
                    comp.wire_list.append(a_conn)
                self.canvas.redraw()

```

UI_Lib/settings_frame.py

```

import customtkinter as ctk
from PIL import Image

class SettingsFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)

```

```

    self.parent = parent
    self.canvas = canvas

    self.menu_on = False

    self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

    def grid_switch_event():
        if canvas.grid.grid_visible:
            self.canvas.grid.grid_visible = False
        else:
            self.canvas.grid.grid_visible = True
        self.canvas.redraw()

        switch_var = ctk.StringVar(value="on")
        switch = ctk.CTkSwitch(self.menu_frame, text="Grid",
command=grid_switch_event,
                               variable=switch_var, onvalue="on",
                               offvalue="off")
        switch.pack(padx=5, pady=5)

        grid_size_label = ctk.CTkLabel(self.menu_frame, text="Grid Size", font=\
("Helvetica", 10), height=20)
        grid_size_label.pack(padx=5, pady=5, anchor="w")

    def optionmenu_callback(choice):
        self.canvas.grid.grid_size = int(choice)
        self.canvas.redraw()

        optionmenu = ctk.CTkOptionMenu(self.menu_frame, values=[ "5", "10",
"20", "30", "40", "50"],
                                         command=optionmenu_callback)
        optionmenu.pack(padx=5, pady=5)
        optionmenu.set("10")

        grid_snap_label = ctk.CTkLabel(self.menu_frame, text="Snap Size", font=\
("Helvetica", 10), height=20)
        grid_snap_label.pack(padx=5, pady=5, anchor="w")

    def snap_option_callback(choice):
        if choice == "Grid Size":
            self.canvas.grid.grid_snap = canvas.grid.grid_size
        else:
            self.canvas.grid.grid_snap = int(choice)
        canvas.redraw()

```

```

        snap_option = ctk.CTkOptionMenu(self.menu_frame, values=["Grid Size",
"5", "10", "20", "30", "40", "50"],
                                         command=snap_option_callback)
        snap_option.pack(padx=5, pady=5)
        snap_option.set("Grid Size")

        self.appearance_mode_label = ctk.CTkLabel(self.menu_frame,
text="Appearance Mode:", anchor="w")
        self.appearance_mode_label.pack(padx=5, pady=5)
        self.appearance_mode_optionmenu = ctk.CTkOptionMenu(self.menu_frame,
                                         values=
["Light", "Dark", "System"],
                                         command=self.change_appearance_mode_event)
        self.appearance_mode_optionmenu.pack(padx=5, pady=5)
        self.appearance_mode_optionmenu.set("Dark")

        my_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/DiagramEditor/icons/settings.png"),
                               dark_image=Image.open
("D:/EETools/DiagramEditor/icons/settings.png"),
                               size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

    @staticmethod
    def change_appearance_mode_event(new_appearance_mode: str):
        ctk.set_appearance_mode(new_appearance_mode)

```

```
import customtkinter as ctk
from PIL import Image

class SettingsFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

    def grid_switch_event():
        if canvas.grid.grid_visible:
            self.canvas.grid.grid_visible = False
        else:
            self.canvas.grid.grid_visible = True
            self.canvas.redraw()

        switch_var = ctk.StringVar(value="on")
        switch = ctk.CTkSwitch(self.menu_frame, text="Grid",
                               command=grid_switch_event,
                               variable=switch_var, onvalue="on",
                               offvalue="off")
        switch.pack(padx=5, pady=5)

        grid_size_label = ctk.CTkLabel(self.menu_frame, text="Grid Size", font=\
("Helvetica", 10), height=20)
        grid_size_label.pack(padx=5, pady=5, anchor="w")

    def optionmenu_callback(choice):
        self.canvas.grid.grid_size = int(choice)
        self.canvas.redraw()

        optionmenu = ctk.CTkOptionMenu(self.menu_frame, values=[ "5", "10",
        "20", "30", "40", "50"], command=optionmenu_callback)
        optionmenu.pack(padx=5, pady=5)
        optionmenu.set("10")

        grid_snap_label = ctk.CTkLabel(self.menu_frame, text="Snap Size", font=\
("Helvetica", 10), height=20)
        grid_snap_label.pack(padx=5, pady=5, anchor="w")
```

```

def snap_option_callback(choice):
    if choice == "Grid Size":
        self.canvas.grid.grid_snap = canvas.grid.grid_size
    else:
        self.canvas.grid.grid_snap = int(choice)
    canvas.redraw()

    snap_option = ctk.CTkOptionMenu(self.menu_frame, values=[ "Grid Size",
    "5", "10", "20", "30", "40", "50"], command=snap_option_callback)
    snap_option.pack(padx=5, pady=5)
    snap_option.set("Grid Size")

    self.appearance_mode_label = ctk.CTkLabel(self.menu_frame,
text="Appearance Mode:", anchor="w")
    self.appearance_mode_label.pack(padx=5, pady=5)
    self.appearance_mode_optionmenu = ctk.CTkOptionMenu(self.menu_frame,
values=
["Light", "Dark", "System"],
command=self.change_appearance_mode_event)
    self.appearance_mode_optionmenu.pack(padx=5, pady=5)
    self.appearance_mode_optionmenu.set("Dark")

    my_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/DiagramEditor/icons/settings.png"),
dark_image=Image.open
("D:/EETools/DiagramEditor/icons/settings.png"),
size=(24, 24))

    button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
    button.pack(side=ctk.LEFT, padx=5, pady=10)

def show_menu(self):
    if not self.menu_on:
        self.menu_frame.place(x=15, y=60)
        self.menu_frame.tkraise()
        self.menu_on = True
    else:
        self.menu_frame.place_forget()
        self.menu_on = False

@staticmethod

```

```
def change_appearance_mode_event(new_appearance_mode: str):
    ctk.set_appearance_mode(new_appearance_mode)
```

UI_Lib/top_frame.py

```
import customtkinter as ctk
from tktooltip import ToolTip
from pathlib import Path
from PIL import Image

import matplotlib.pyplot as plt
import PySpice.Logging.Logging as Logging
logger = Logging.setup_logging()
from PySpice.Doc.ExampleTools import find_libraries
from PySpice.Spice.Library import SpiceLibrary
from PySpice.Spice.Netlist import Circuit

from UI_Lib.file_menu_frame import FileMenuFrame
from UI_Lib.settings_frame import SettingsFrame
from UI_Lib.help_frame import HelpFrame
from Wire_Lib import AnalogWire


class TopFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):
        super().__init__(parent)
        self.parent = parent
        self.canvas = canvas

        # Add Top Frame widget here
        file_frame = FileMenuFrame(self.parent, self, self.canvas)
        file_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        settings_frame = SettingsFrame(self.parent, self, self.canvas)
        settings_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        help_frame = HelpFrame(self.parent, self, self.canvas)
        help_frame.pack(side=ctk.RIGHT, padx=5, pady=5)

        a_image = ctk.CTkImage(light_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
                               dark_image=Image.open(Path(__file__).parent /
"../icons/angle.png"),
```

```

        size=(24, 24))
    self.button_id = ctk.CTkButton(self, text="", image=a_image, width=30,
command=self.rotate_comp)
    self.button_id.pack(side=ctk.LEFT, padx=5, pady=5)
    ToolTip(self.button_id, msg="Rotate selected component")

    a_image = ctk.CTkImage(light_image=Image.open(Path(__file__).parent /
"../icons/analyze.png"),
                           dark_image=Image.open(Path(__file__).parent /
"../icons/analyze.png"),
                           size=(24, 24))
    self.button_id = ctk.CTkButton(self, text="", image=a_image, width=30,
command=self.analyze_circuit)
    self.button_id.pack(side=ctk.LEFT, padx=5, pady=5)
    ToolTip(self.button_id, msg="Analyze circuit")

def rotate_comp(self):
    self.parent.rotate_comp(_event=None)

def analyze_circuit(self):
    libraries_path = find_libraries()
    _spice_library = SpiceLibrary(libraries_path)

    units = {
        'u_V': 1,
        'u_kΩ': 1000,
        'u_kHz': 1000,
        'u_uF': 1e-6,
        'u_MΩ': 1e6,
        'u_pF': 1e-12
    }

    freq_amplitude = 1
    freq_units = units['u_kHz']

    source = None

    circuit = Circuit('Transistor')

    for comp in self.canvas.comp_list:
        if not isinstance(comp, AnalogWire):
            if comp.comp_type == 'dc_source':
                node_list = self.get_nodes(comp)
                circuit.V('power', node_list[0], circuit.gnd, comp.value *
units[comp.units])
            elif comp.comp_type == 'ac_source':

```

```

        node_list = self.get_nodes(comp)
        source = circuit.SinusoidalVoltageSource('in',
node_list[0], circuit.gnd,
                                amplitude=comp.value*units[comp.units],
frequency=freq_amplitude*freq_units)
        elif comp.comp_type == 'resistor':
            comp_num = self.get_comp_num(comp.text)
            node_list = self.get_nodes(comp)
            circuit.R(comp_num, node_list[0], node_list[1],
comp.value*units[comp.units])
        elif comp.comp_type == 'capacitor':
            comp_num = self.get_comp_num(comp.text)
            node_list = self.get_nodes(comp)
            circuit.C(comp_num, node_list[0], node_list[1], comp.value
* units[comp.units])
        elif comp.comp_type == 'inductor':
            comp_num = self.get_comp_num(comp.text)
            node_list = self.get_nodes(comp)
            circuit.L(comp_num, node_list[0], node_list[1], comp.value
* units[comp.units])
        elif comp.comp_type == 'npn_transistor':
            comp_num = self.get_comp_num(comp.text)
            node_list = self.get_nodes(comp)
            circuit.BJT(comp_num, node_list[1], node_list[0],
node_list[2], model='bjt')
            circuit.model('bjt', 'npn', bf=comp.bf,
cjc=comp.cjc*units[comp.cjc_units], rb=comp.rb)

        print(circuit)
# Create circuit plots
figure, ax = plt.subplots(figsize=(20, 10))

# .ac dec 5 10m 1G

simulator = circuit.simulator(temperature=25, nominal_temperature=25)
analysis = simulator.transient(step_time=source.period / 200,
end_time=source.period * 2)

ax.set_title(' ')
ax.set_xlabel('Time [s]')
ax.set_ylabel('Voltage [V]')
ax.grid()
ax.plot(analysis['in'])
ax.plot(analysis.out)
ax.legend(['input', 'output'], loc=(.05, .1))

```

```
plt.tight_layout()
plt.show()

@staticmethod
def get_comp_num(comp_text):
    comp_num = comp_text[1:]
    return comp_num

def get_nodes(self, in_comp):
    node_list = []
    for conn in in_comp.wire_list:
        for comp in self.canvas.comp_list:
            if isinstance(comp, AnalogWire) and comp.name == conn.wire_name:
                node_list.append(comp.node_num)
    return node_list
```

UI_Lib/wire_button_frame.py

```
    "../icons/straight_line.png"),
            dark_image=Image.open
                (Path(__file__).parent /
 "../icons/straight_line.png"),
            size=(24, 24))

    straight_wire_button = ctk.CTkButton(self, text="",
image=straight_wire_image, width=30,
                                         command=self.create_straight_wire)
    straight_wire_button.grid(row=1, column=0, sticky=ctk.W, padx=2,
pady=2)
    ToolTip(straight_wire_button, msg='Straight wire')

    segment_wire_image = ctk.CTkImage(light_image=Image.open
                                         (Path(__file__).parent /
 "../icons/segment_line.png"),
            dark_image=Image.open
                (Path(__file__).parent /
 "../icons/segment_line.png"),
            size=(24, 24))

    segment_wire_button = ctk.CTkButton(self, text="",
image=segment_wire_image, width=30,
                                         command=self.create_segment_wire)
    segment_wire_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
    ToolTip(segment_wire_button, msg='Segment wire')

    elbow_wire_image = ctk.CTkImage(light_image=Image.open
                                         (Path(__file__).parent /
 "../icons/elbow_line.png"),
            dark_image=Image.open
                (Path(__file__).parent /
 "../icons/elbow_line.png"),
            size=(24, 24))

    elbow_wire_button = ctk.CTkButton(self, text="",
image=elbow_wire_image, width=30,
                                         command=self.create_elbow_wire)
    elbow_wire_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)
    ToolTip(elbow_wire_button, msg='Elbow wire')

    node_image = ctk.CTkImage(light_image=Image.open
                                         (Path(__file__).parent / "../icons/node.png"),
            dark_image=Image.open
                (Path(__file__).parent / "../icons/node.png"),
            size=(24, 24))
```

```

node_button = ctk.CTkButton(self, text="", image=node_image, width=30,
                           command=self.create_node)
node_button.grid(row=2, column=0, sticky=ctk.W, padx=2, pady=2)
ToolTip(node_button, msg='Node')

# Shape button handlers
def create_straight_wire(self):
    wire = AnalogWire(self.canvas, 'straight', 0, 0, 0, 0)
    self.create_wire(wire)

def create_segment_wire(self):
    wire = AnalogWire(self.canvas, 'segment', 0, 0, 0, 0)
    self.create_wire(wire)

def create_elbow_wire(self):
    wire = AnalogWire(self.canvas, 'elbow', 0, 0, 0, 0)
    self.create_wire(wire)

def create_node(self):
    node = Node(self.canvas, 100, 100)
    self.canvas.comp_list.append(node)
    self.canvas.redraw()
    self.canvas.mouse.move_mouse_bind_events()

def create_wire(self, wire):
    self.assign_wire_name(wire)
    self.canvas.mouse.current_wire_obj = wire
    self.canvas.show_connectors()
    self.canvas.comp_list.append(wire)
    self.canvas.mouse.draw_wire_mouse_events()

def assign_wire_name(self, wire):
    self.wire_count += 1
    wire_name = 'wire' + str(self.wire_count)
    wire.name = wire_name
    self.canvas.wire_dict[wire_name] = wire

```

Wire_Lib/__init__.py

```

# import wire related classes
from Wire_Lib.analog_wire import AnalogWire
from Wire_Lib.node import Node

```

```
# import wire selector related classes
from Wire_Lib.wire_selector import WireSelector
```

Wire_Lib/wire.py

```
from Wire_Lib.wire_selector import WireSelector

class Wire:
    """Base class for wire classes"""
    def __init__(self, canvas, wire_type, x1, y1, x2, y2):
        self.canvas = canvas
        self.wire_type = wire_type
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

        self.fill_color = "black"
        self.border_width = 2
        self.wire_dir = self.canvas.wire_dir

        self.name = None
        self.id = None
        self.is_selected = False
        self.selector = None
        self.width = 2
        self.bbox = None
        self.node_num = '0'

    # Connections for wire list
    self.in_cnx = None
    self.out_cnx = None
    self.cnx = []

    self.sel1, self.sel2 = None, None

    def update_bbox(self):
        self.bbox = self.canvas.bbox(self.id)

    def create_selectors(self):
        self.sel1 = WireSelector(self.canvas, "begin", self.x1, self.y1)
```

```

        self.sel2 = WireSelector(self.canvas, "end", self.x2, self.y2)

    def update_selectors(self):
        self.sel1.x, self.sel1.y = self.x1, self.y1
        self.sel2.x, self.sel2.y = self.x2, self.y2
        self.sel1.update()
        self.sel2.update()

    def update_selection(self):
        if self.is_selected:
            self.canvas.itemconfigure(self.id, fill="red")
            self.canvas.itemconfigure(self.sel1.id, state='normal')
            self.canvas.itemconfigure(self.sel2.id, state='normal')
        else:
            self.canvas.itemconfigure(self.id, fill="black")
            self.canvas.itemconfigure(self.sel1.id, state='hidden')
            self.canvas.itemconfigure(self.sel2.id, state='hidden')

    def hit_test(self, x, y):
        x1, y1 = self.bbox[0], self.bbox[1]
        x2, y2 = self.bbox[2], self.bbox[3]
        if x1 <= x <= x2 and y1 <= y <= y2:
            self.is_selected = True
        else:
            self.is_selected = False

    def sel_hit_test(self, x, y):
        if self.sel1.selector_hit_test(x, y):
            self.selector = self.sel1.name
            return self.sel1
        elif self.sel2.selector_hit_test(x, y):
            self.selector = self.sel2.name
            return self.sel2
        else:
            return None

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
            self.x2, self.y2 = self.canvas.grid.snap_to_grid(self.x2, self.y2)
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1

```

```
    self.x1, self.y1 = x1, y1
    self.x1, self.y1 = self.canvas.grid.snap_to_grid(self.x1, self.y1)

def set_node_num(self, node_num):
    self.node_num = node_num
```

Wire_Lib/analog_wire.py

```

tags="wire")
    elif self.wire_dir == "V": # Vertical
        self.id = self.canvas.create_line(self.x1, self.y1, self.x1,
self.y2,
                                         self.x1, self.y2, self.x2,
self.y2,
                                         fill=self.fill_color,
                                         width=self.border_width,
tags="wire")

def create_segment_wire(self):
    w = self.x2 - self.x1
    h = self.y2 - self.y1

    if self.wire_dir == "H": # Horizontal
        self.seg1 = self.x1, self.y1, self.x1 + w / 2, self.y1
        self.seg2 = self.x1 + w / 2, self.y1, self.x1 + w / 2, self.y2
        self.seg3 = self.x1 + w / 2, self.y2, self.x2, self.y2
    elif self.wire_dir == "V": # Vertical
        self.seg1 = self.x1, self.y1, self.x1, self.y1 + h / 2
        self.seg2 = self.x1, self.y1 + h / 2, self.x2, self.y1 + h / 2
        self.seg3 = self.x2, self.y1 + h / 2, self.x2, self.y2
    self.segment_list = [self.seg1, self.seg2, self.seg3]
    self.draw_segments()

def draw_segments(self):
    for s in self.segment_list:
        self.id = self.canvas.create_line(s, fill=self.fill_color,
width=self.border_width, tags='wire')

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_selectors()
    self.update_selection()

def update_position(self):
    if self.wire_type == 'straight':
        self.update_straight_position()
    elif self.wire_type == 'elbow':
        self.update_elbow_position()
    elif self.wire_type == 'segment':
        self.update_segment_position()

def update_straight_position(self):
    """Update the position when the attached component is moved"""

```

```

        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)

    def update_elbow_position(self):
        """Update the position when the attached component is moved"""
        if self.wire_dir == "H":
            self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y1,
                               self.x2, self.y1, self.x2, self.y2)
        elif self.wire_dir == "V":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y2,
                               self.x1, self.y2, self.x2, self.y2)

    def update_segment_position(self):
        """Update the position when the attached component is moved"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        if self.wire_dir == "H":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1 + w / 2,
                               self.y1,
                               self.x1 + w / 2, self.y1, self.x1 + w / 2,
                               self.y2,
                               self.x1 + w / 2, self.y2, self.x2, self.y2)
        elif self.wire_dir == "V":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y1 + h
/ 2,
                               self.x1, self.y1 + h / 2, self.x2, self.y1 + h /
2,
                               self.x2, self.y1 + h / 2, self.x2, self.y2)

    def hit_test(self, x, y):
        # 2-Point Line equation: y = m * (x - x1) + y1
        x1, y1 = self.x1, self.y1
        x2, y2 = self.x2, self.y2

        # Calculate the slope: m = (y2 - y1) / (x2 - x1)
        if (x2 - x1) == 0:
            m = 0
        else:
            m = (y2 - y1)/(x2 - x1)

        # Check to see if the point (x, y) is on the line and between the two
        end points
        tol = 10
        if y - tol <= m*(x - x1) + y1 <= y + tol:
            if (min(x1, x2) <= x <= max(x1, x2)) and (min(y1, y2) <= y <
max(y1, y2)):
                self.is_selected = True

```

```

        else:
            self.is_selected = False

    def __repr__(self):
        return ("Type: " + self.wire_type + " node_num: " + self.node_num +
               " x1: " + str(self.x1) + " y1: " + str(self.y1) +
               " x2: " + str(self.x2) + " y2: " + str(self.y2))

    def reprJson(self):
        return dict(type=self.wire_type, wire_dir=self.wire_dir,
node_num=self.node_num, x1=self.x1, y1=self.y1,
               x2=self.x2, y2=self.y2, name=self.name)

```

Wire_Lib/node.py

```

from Helper_Lib import Point
from Comp_Lib import Connector

class Node:
    def __init__(self, canvas, x1, y1):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1

        self.text = '0'
        self.comp_type = 'node'
        self.id = None
        self.text_id = None
        self.sel_id = None
        self.wire_list = []
        self.bbox = None
        self.angle = 0
        self.is_selected = False
        self.is_drawing = False
        self.conn = None
        self.conn_list = []

        self.create()

    def create(self):
        self.create_node()
        self.update_bbox()

```

```

        self.create_text()
        self.create_selector()
        self.create_connectors()
        self.set_connector_visibility()

    def create_node(self):
        self.id = self.canvas.create_oval(self.x1 - 5, self.y1 - 5, self.x1 +
5, self.y1 + 5, fill="black")

    def update_bbox(self):
        """Update the bounding box to get current gate coordinates"""
        self.bbox = self.canvas.bbox(self.id)

    def create_text(self):
        text_loc = Point(self.x1-10, self.y1-10) # Put text above symbol
        self.text_id = self.canvas.create_text(text_loc.x, text_loc.y,
                                              text=self.text, fill="black",
                                              font='Helvetica 10 bold',
                                              angle=self.angle, tags="text")

    def create_selector(self):
        """Create the red rectangle selector and check to see if the gate is
selected"""
        x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
        self.sel_id = self.canvas.create_rectangle(x1, y1, x2, y2, fill=None,
outline="red", width=2)
        self.set_selector_visibility()

    def set_selector_visibility(self):
        """Set the selector visibility state"""
        if self.is_selected:
            self.canvas.itemconfig(self.sel_id, state='normal')
        else:
            self.canvas.itemconfig(self.sel_id, state='hidden')

    def create_connectors(self):
        # Calculate position of connectors from current comp position and size
        center = self.get_geometry()

        self.conn = Connector(self.canvas, "conn", center.x, center.y)
        self.conn_list = [self.conn]

    def get_geometry(self):
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1

```

```

        h = y2 - y1
        center = Point(x1 + w / 2, y1 + h / 2)

    return center

def set_connector_visibility(self):
    """Set the connector visibility state"""
    if self.is_drawing:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='normal')
    else:
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

def update(self):
    self.update_position()
    self.update_bbox()
    self.update_text()
    self.update_selector()
    self.update_connectors()
    self.set_connector_visibility()

def update_position(self):
    self.canvas.coords(self.id, self.x1 - 5, self.y1 - 5, self.x1 + 5,
self.y1 + 5) # Update position

def update_text(self):
    self.canvas.itemconfig(self.text_id, text=self.text)
    self.canvas.coords(self.text_id, self.x1 - 10, self.y1 - 10)

def update_selector(self):
    """Update the red rectangle selector coordinates and check to see if
the gate is selected"""
    x1, y1, x2, y2 = self.bbox[0] - 5, self.bbox[1] - 5, self.bbox[2] + 5,
self.bbox[3] + 5
    self.canvas.coords(self.sel_id, x1, y1, x2, y2)
    self.set_selector_visibility()

def update_connectors(self):
    """Update the position of connectors here"""
    center = self.get_geometry()

    self.conn.x, self.conn.y = center.x, center.y

    for c in self.conn_list:
        c.update()

```

```

        self.move_connected_wires()

    def check_connector_hit(self, x, y):
        """Hit test to see if a connector is at the provided x, y
coordinates"""
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None

    def move_connected_wires(self):
        """Resize connected wires if the shape is moved"""
        for connection in self.wire_list: # comp_conn, wire_name, wire_end
            for connector in self.conn_list:
                if connector.name == connection.comp_conn:
                    wire_obj = self.canvas.wire_dict[connection.wire_name]
                    if connection.wire_end == "begin":
                        wire_obj.x1 = connector.x
                        wire_obj.y1 = connector.y
                    elif connection.wire_end == "end":
                        wire_obj.x2 = connector.x
                        wire_obj.y2 = connector.y

    def rotate(self):
        """Set the rotation angle to the current angle + 90 deg, reset to 0 deg
if angle > 270 deg"""
        self.angle += 90
        if self.angle > 270:
            self.angle = 0

    def hit_test(self, x, y):
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        if x1 <= x <= x2 and y1 <= y <= y2:
            self.is_selected = True
        else:
            self.is_selected = False

    def __repr__(self):
        return ("Type: " + self.comp_type + " Text: " + self.text + " x1: " +
str(self.x1) + " y1: " +
str(self.y1) + " wire list: " + str(self.wire_list.__repr__()))

    def reprJson(self):
        return dict(type=self.comp_type, text=self.text, x1=self.x1,
y1=self.y1, angle=self.angle,

```

```
        wire_list=self.wire_list)
```

Wire_Lib/wire_selector.py

```
class WireSelector:
    def __init__(self, canvas, name, x, y):
        self.canvas = canvas
        self.name = name
        self.x = x
        self.y = y
        self.is_selected = False

        self.id = self.canvas.create_oval(self.x - 5, self.y - 5, self.x + 5,
self.y + 5,
                                         state='normal', fill="white")

    def update(self):
        self.update_position()
        self.update_selection()

    def update_position(self):
        """Update the selector position"""
        sel_points = [self.x - 5, self.y - 5, self.x + 5, self.y + 5]
        self.canvas.coords(self.id, sel_points)

    def update_selection(self):
        if self.is_selected:
            self.canvas.itemconfigure(self.id, fill="yellow")
        else:
            self.canvas.itemconfigure(self.id, fill="white")

    def selector_hit_test(self, event_x, event_y):
        if self.x-5 <= event_x <= self.x+5 and self.y-5 <= event_y <= self.y+5:
            self.is_selected = True
            self.update_selection()
            return True
        else:
            self.is_selected = False
            self.update_selection()
            return False
```

analog_simulator.py

```
import customtkinter as ctk
from UI_Lib import LeftFrame, TopFrame, Canvas

ctk.set_appearance_mode("System") # Modes: "System" (standard), "Dark",
"Light"
ctk.set_default_color_theme("blue") # Themes: "blue" (standard), "green",
"dark-blue"

class App(ctk.CTk):
    """ctk.CTk is a CustomTkinter main window, similar to tk.Tk tkinter main
window"""
    def __init__(self):
        super().__init__()
        self.geometry("1200x800x100x100") # w, h, x, y
        self.title("Analog Simulator")

        self.canvas = Canvas(self)
        self.left_frame = LeftFrame(self, self.canvas)
        self.top_frame = TopFrame(self, self.canvas)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)
        self.canvas.pack(side=ctk.LEFT, fill=ctk.BOTH, expand=True)

        # Add bindings here
        self.bind("<Configure>", self.on_window_resize)
        self.bind('<r>', self.rotate_comp)
        self.bind('<h>', self.canvas.set_horiz_dir)
        self.bind('<v>', self.canvas.set_vert_dir)
        self.canvas.bind('<Button-3>', self.canvas.edit_shape)

    def on_window_resize(self, _event):
        self.canvas.redraw()

    def rotate_comp(self, _event=None):
        if self.canvas.mouse.selected_comp:
            self.canvas.mouse.selected_comp.rotate()
            self.canvas.redraw()

if __name__ == "__main__":
    """Instantiate the Microwave Simulator app and run the main loop"""

```

```
app = App()
app.mainloop()
```

Conclusion

This concludes all EE Tools design projects. I hope you enjoyed the journey as much as I did. Ever since I designed my first microwave circuit on Touchstone, I wanted to know how to write electrical engineering software design tools. As you can see, it involves learning a software language such as Python, taking incremental development steps such as the projects in this book:

- Python Tutorial
- Beginner Project - Scientific Calculator
- Intermediate Projects
 - Shape Editor
 - Line Editor
- Advanced Projects
 - Diagram Editor
 - Digital Circuit Simulator
 - Microwave Circuit Simulator
 - Analog Circuit Simulator

There is much room for improvement and the reader is encouraged to modify the code to add features, make it more robust, and explore areas of interest. Good luck in your engineering journey.

R. A. Crist

Appendix: Python References

Websites

- [W3 Schools Python Tutorials](#)
- [TutorialsPoint Python Tutorials](#)
- [Geeks for Geeks Python Tutorials](#)

YouTube Videos

- [Bro Code Python Tutorial for Beginners](#)
- [Programming with Mosh - Python Full Course for Beginners](#)

E-books

- "Python Crash Course", Eric Matthes, No Starch Press, 2016, Kindle edition.
- "Python 3 Object Oriented Programming", Dusty Phillips, PACKT Publishing, 2010, Kindle edition.
- "Learn Python Quickly", Code Quickly, 2022, Kindle edition.

Programming Assistant

- [ChatGPT](#)