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

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

Mouse Features:

- [x] 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

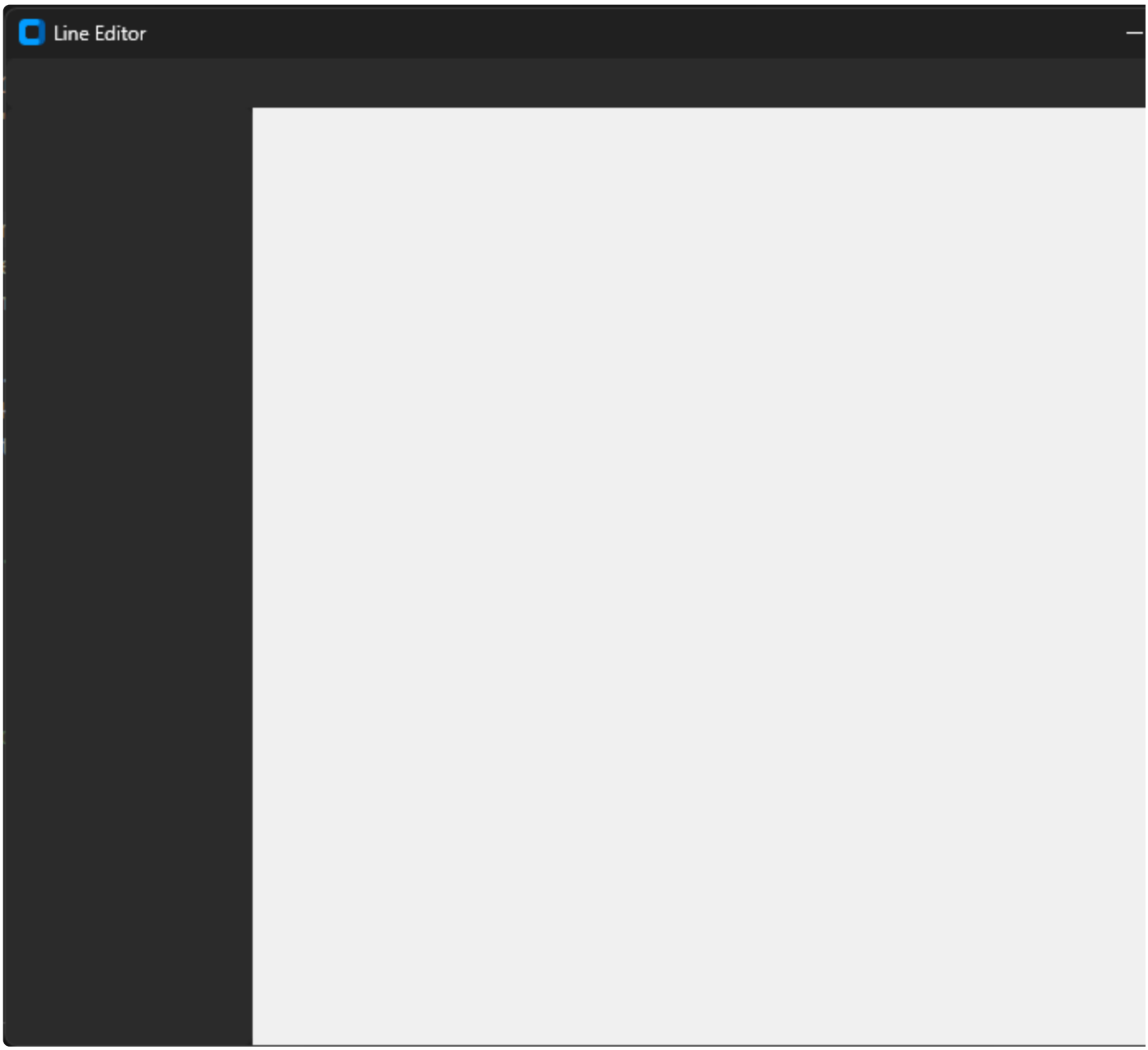
- Create a CustomTkinter window
- Add a Canvas
- Add a Top Frame
- Add a Left Frame

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:

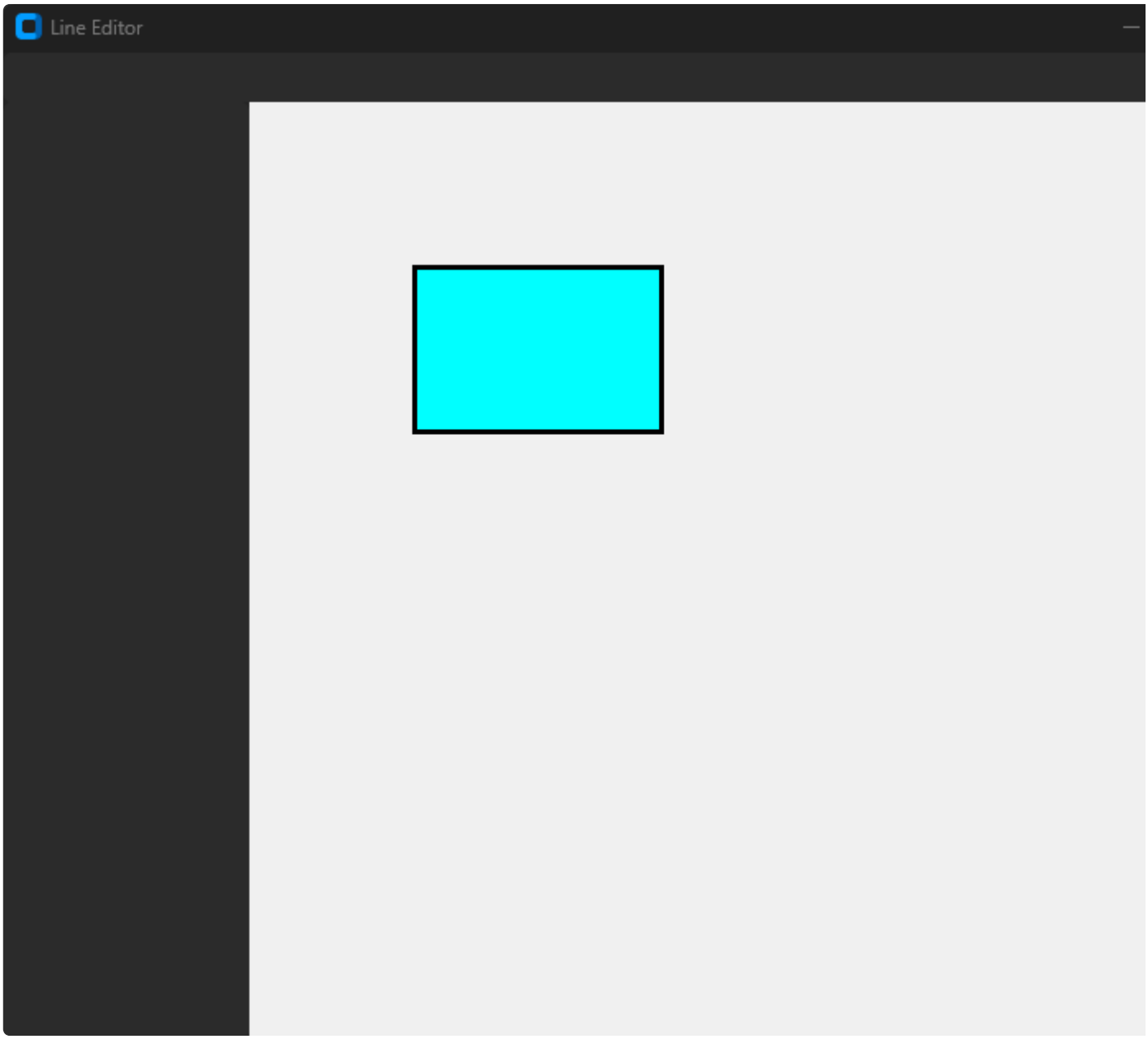
- Create a Rectangle class
- Draw a cyan rectangle on the canvas

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

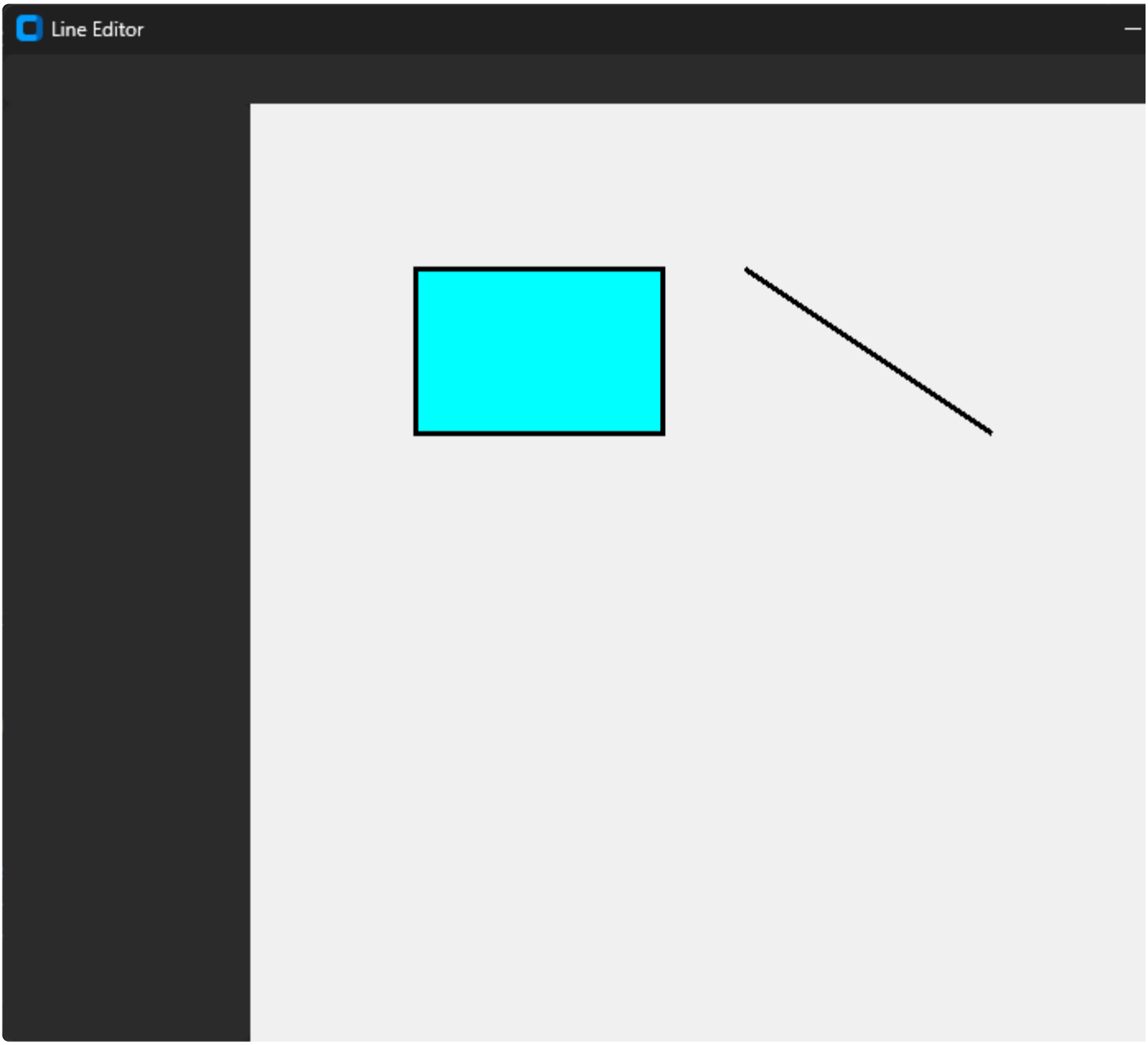
As we did with the rectangle class, add a line class to the Line Editor file above the definition for the Line Editor App class. Create a line object in the Line Editor App class just below the code for creating the rectangle.

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

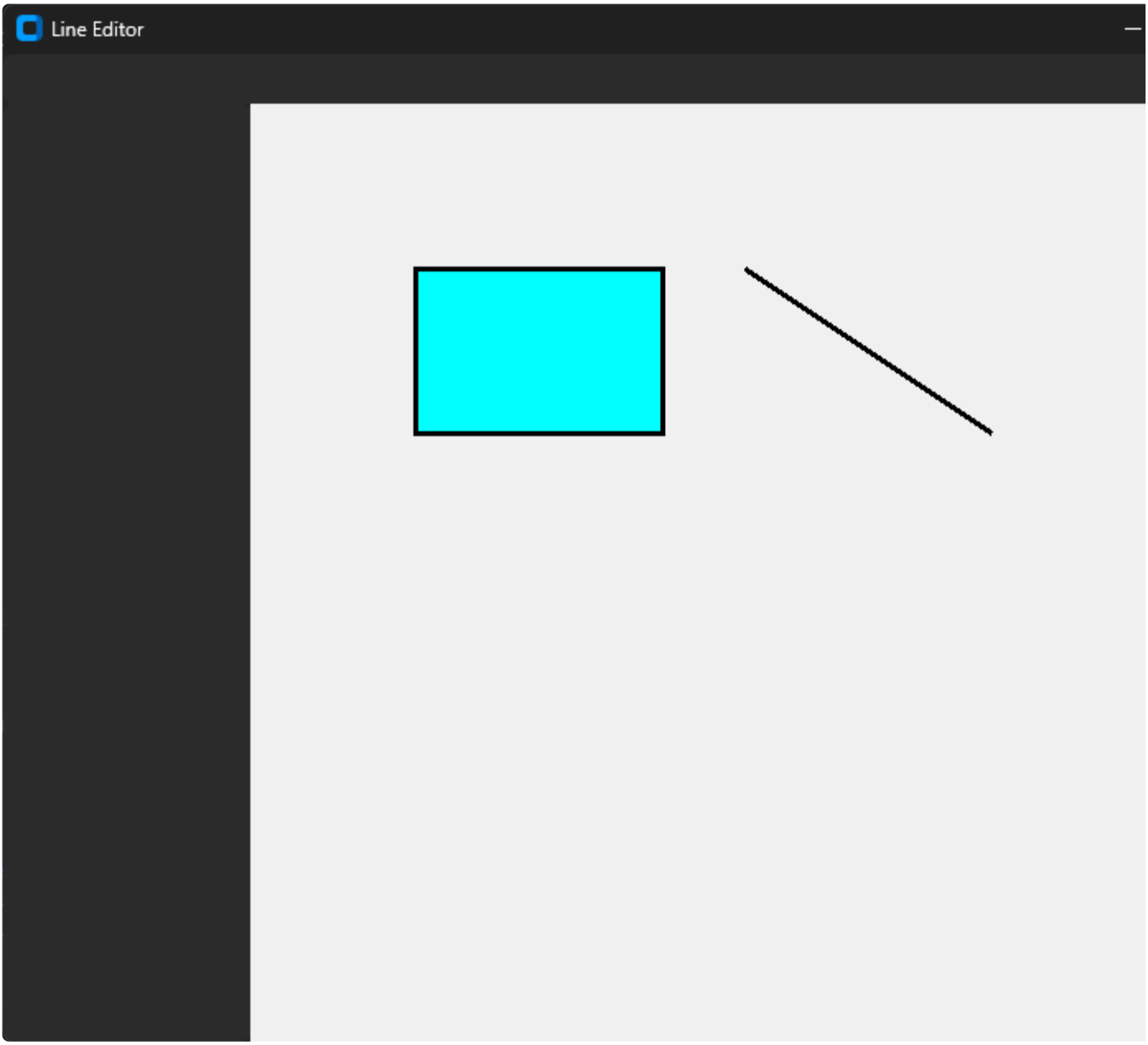
```
class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()
        self.shape_list = []

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 out 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.

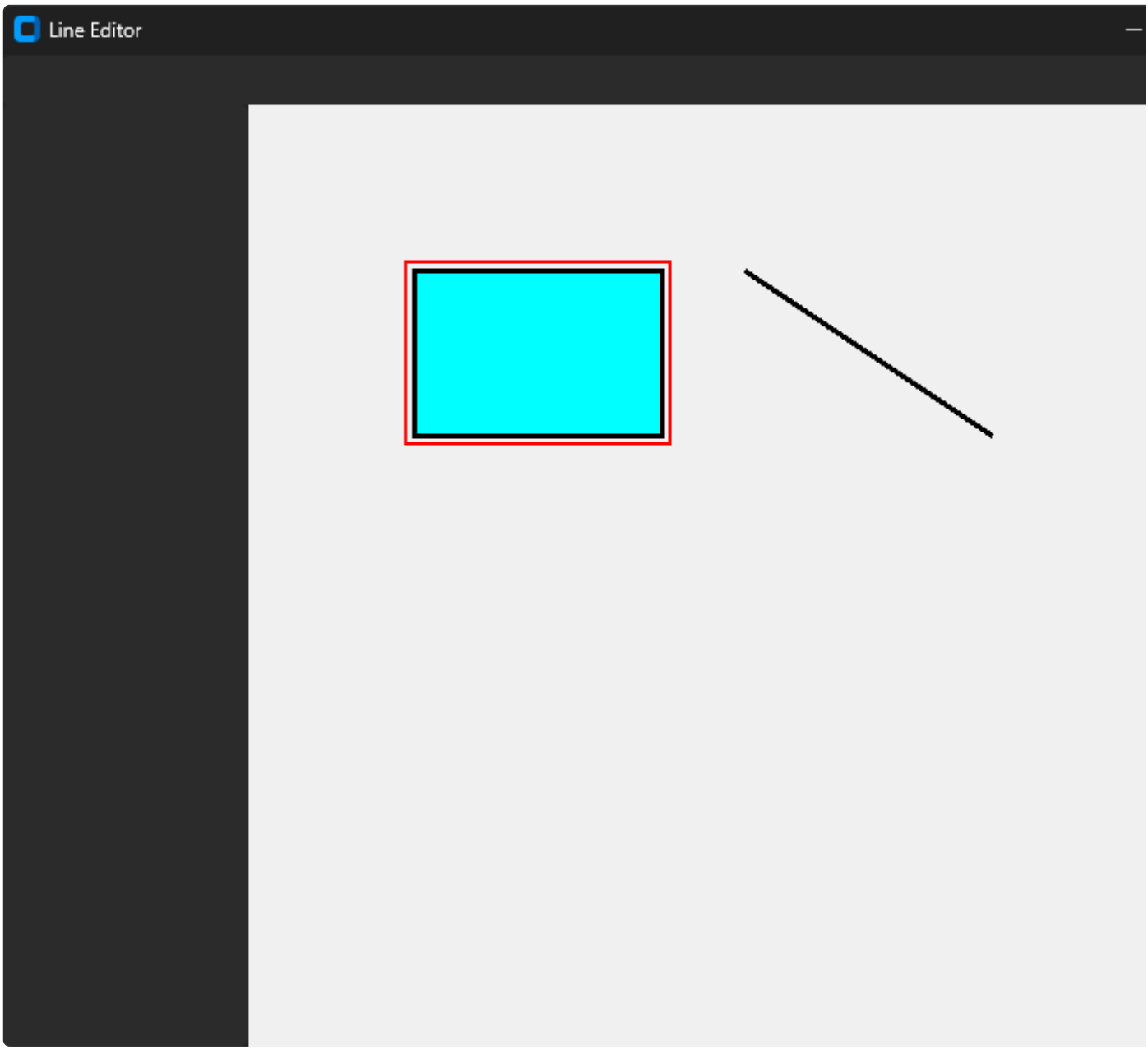
Rectangle Class

```
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)
```

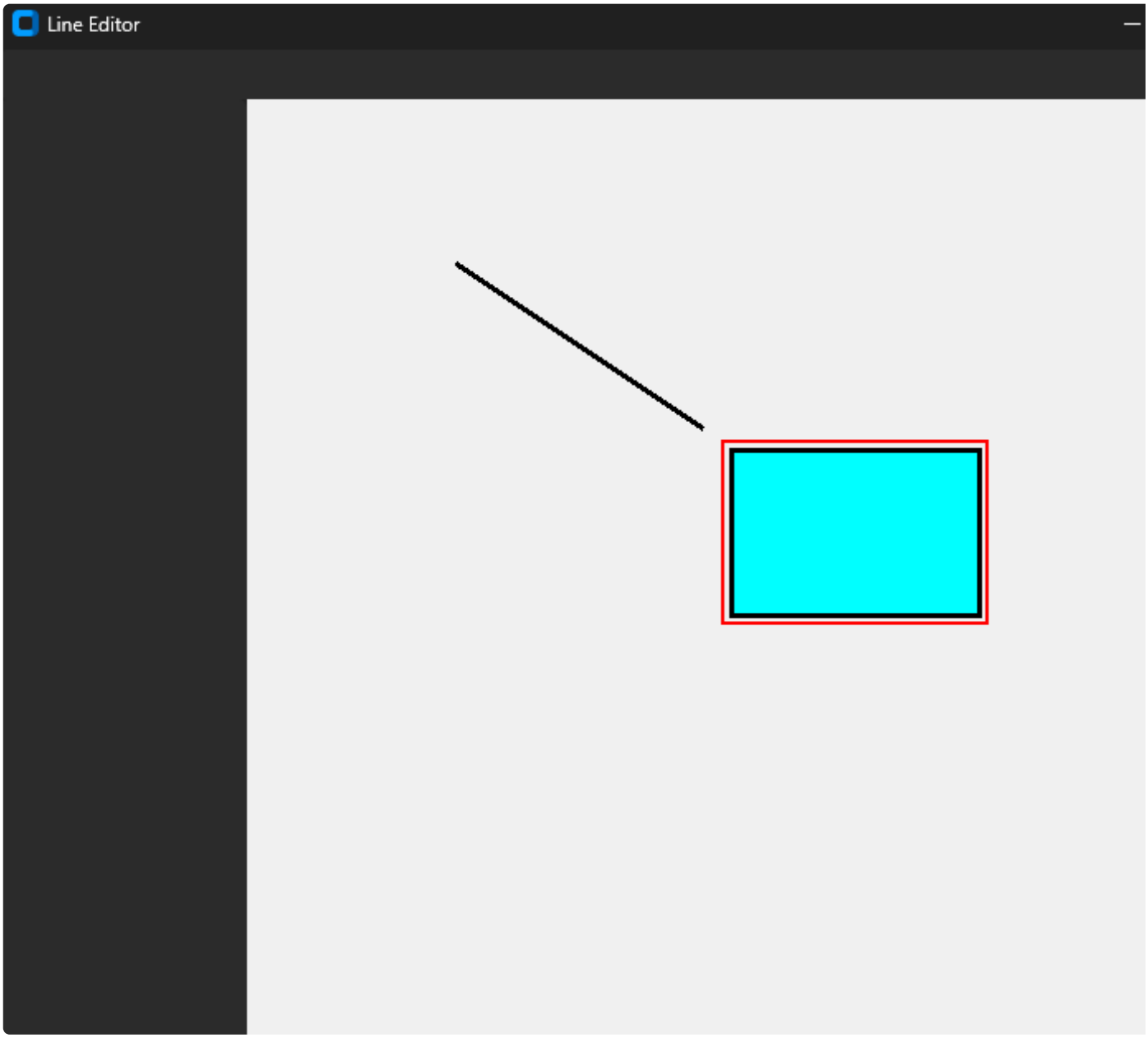
```
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    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 is is draw 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 - h/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.

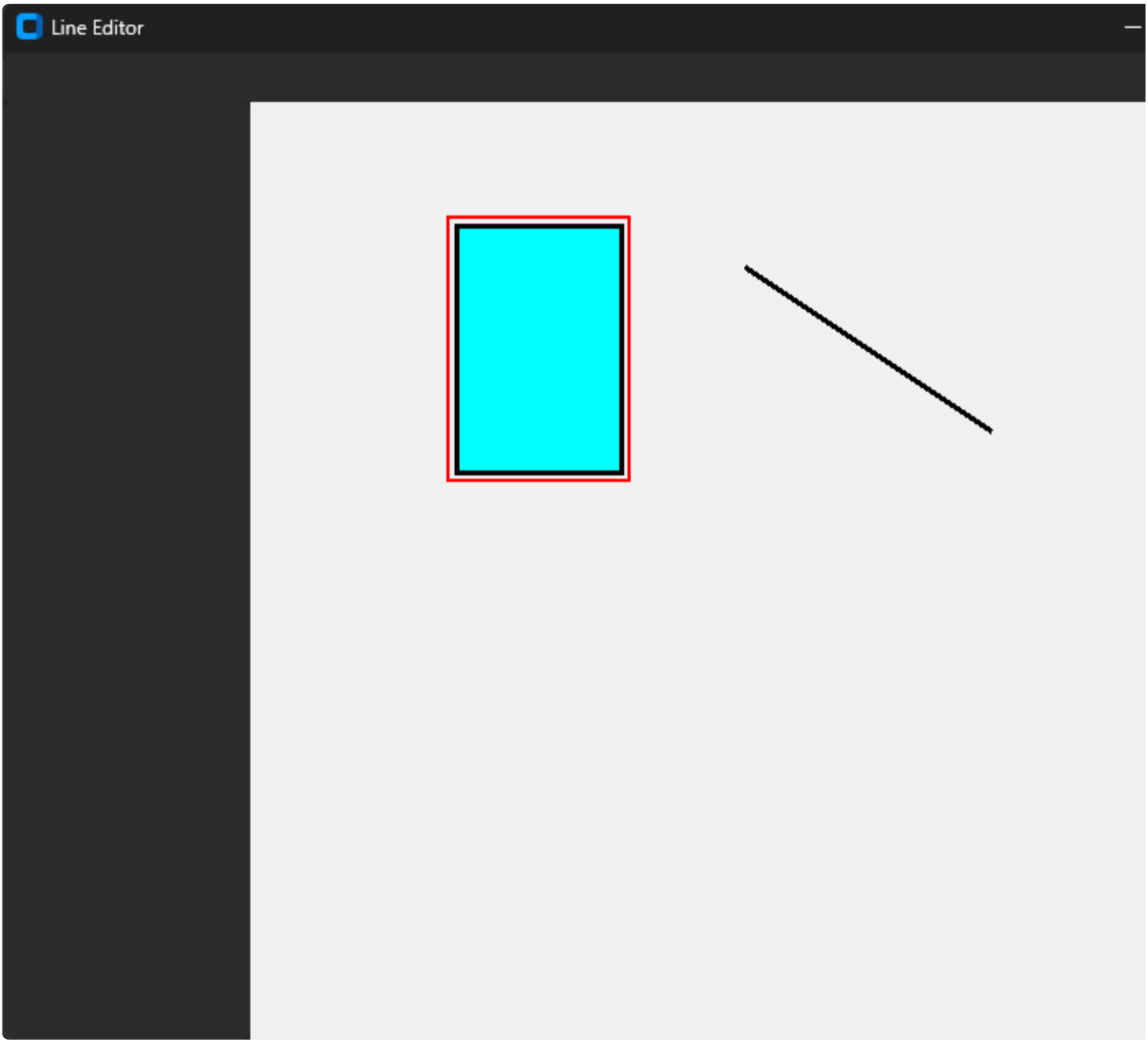
```
class Canvas(ctk.CTkCanvas):
    def __init__(self):
        super().__init__()
        self.shape_list = []
        self.mouse = Mouse(self)
```

```
    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.

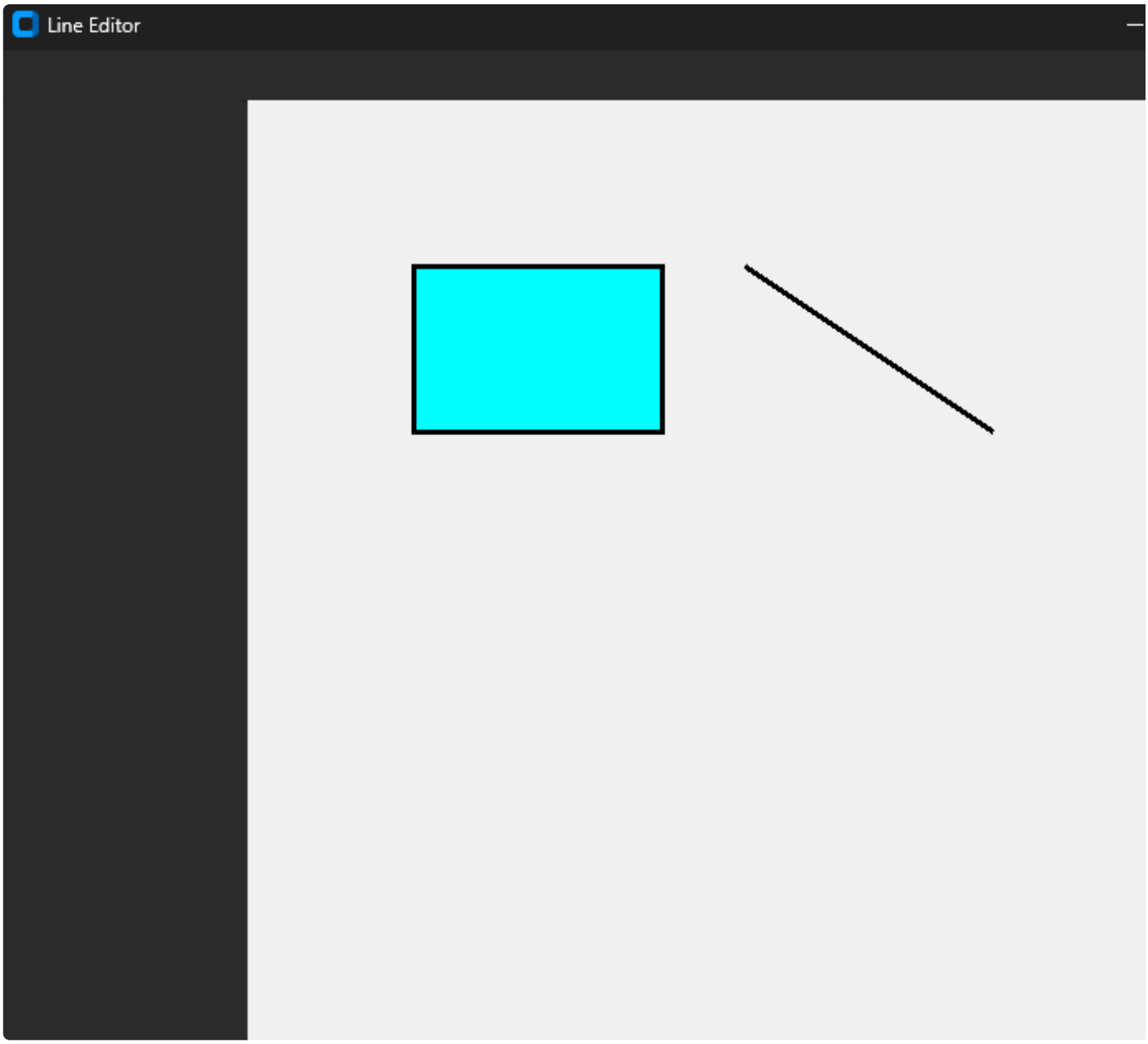
```
class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)

    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)
```

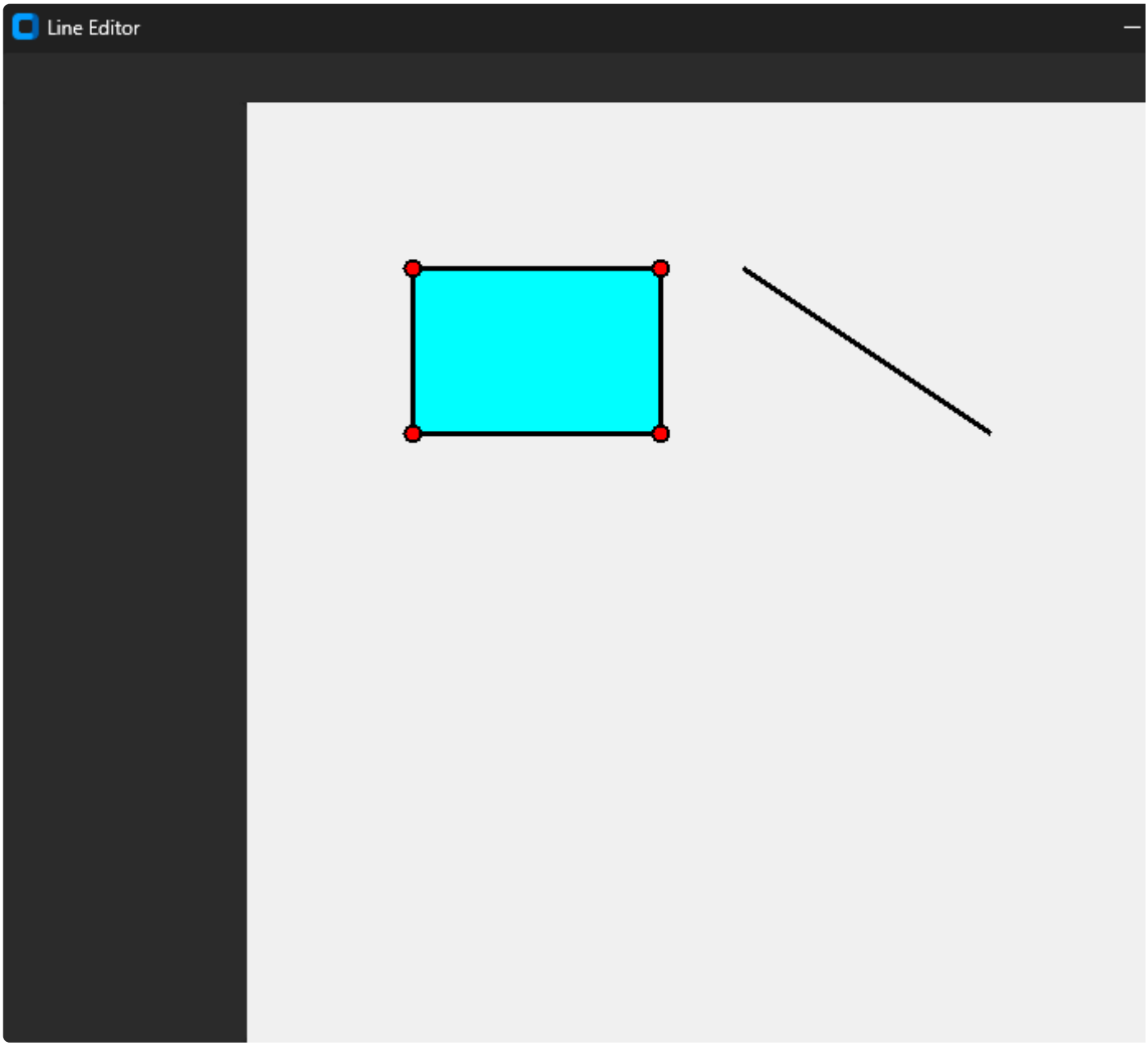
```
        self.s3 = Selector(self.canvas, "s3", x2, y2)
        self.s4 = Selector(self.canvas, "s4", x1, y2)

        # 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 - h/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
```

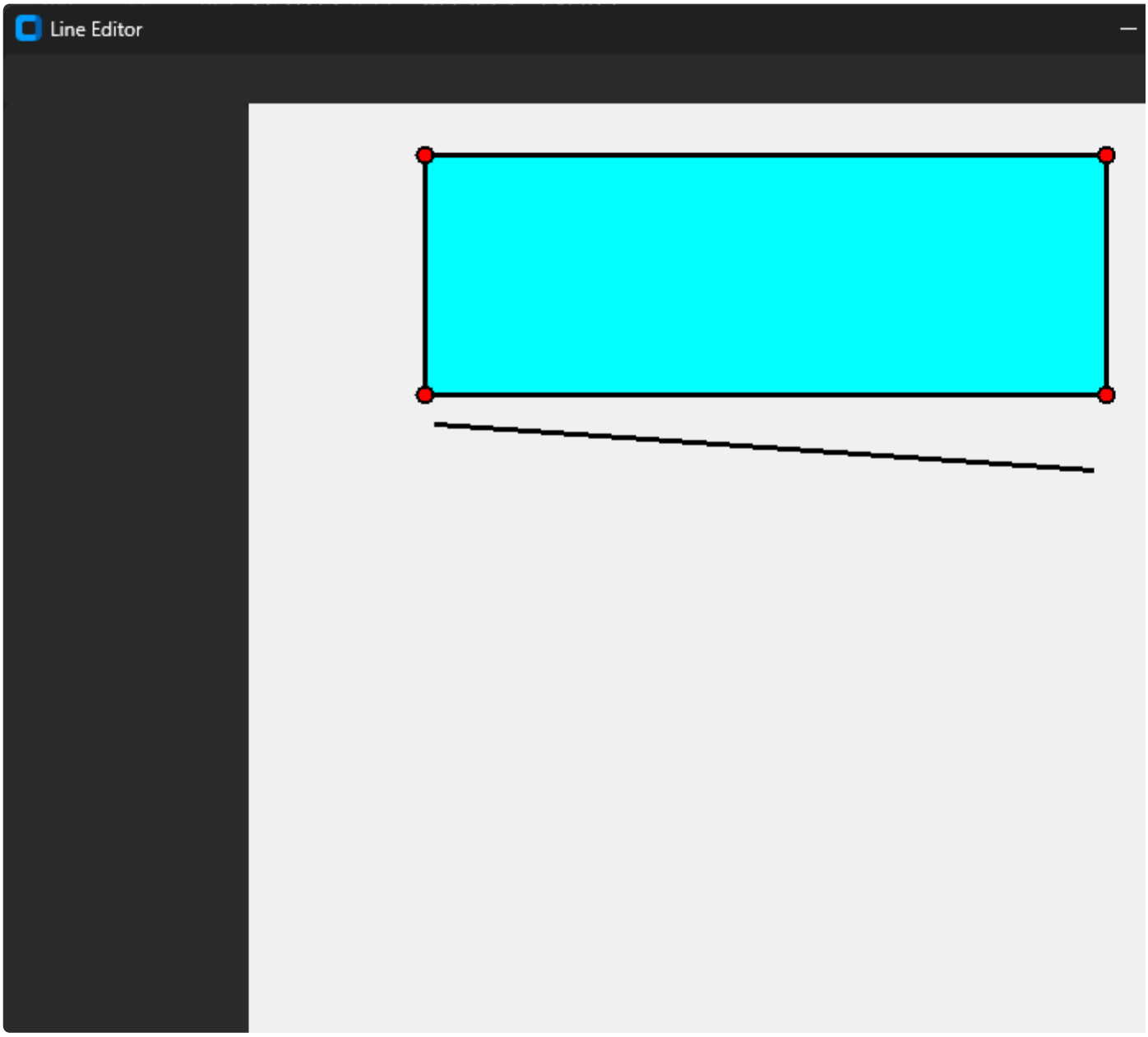
Modify the Mouse Class to add resize bind mouse events method that unbinds all current bindings and binds resize left down, resize left drag, and resize left up methods. The resize left down method calculates the offsets from the current mouse position to the shape coordinates. The resize left drag method 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.

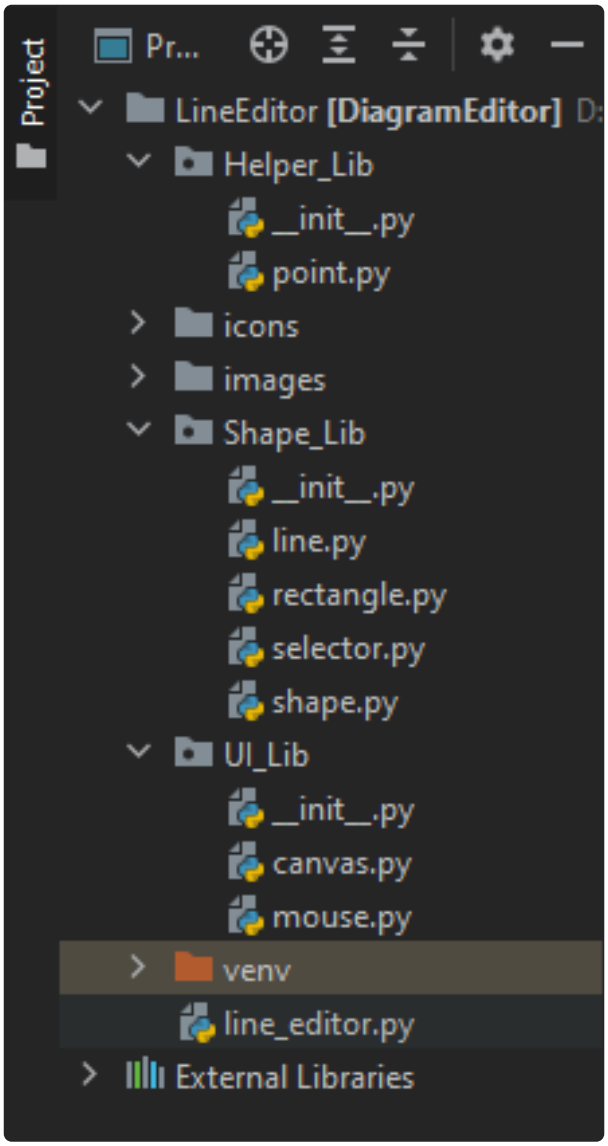
```
from Helper_Lib.point import Point


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 =
```

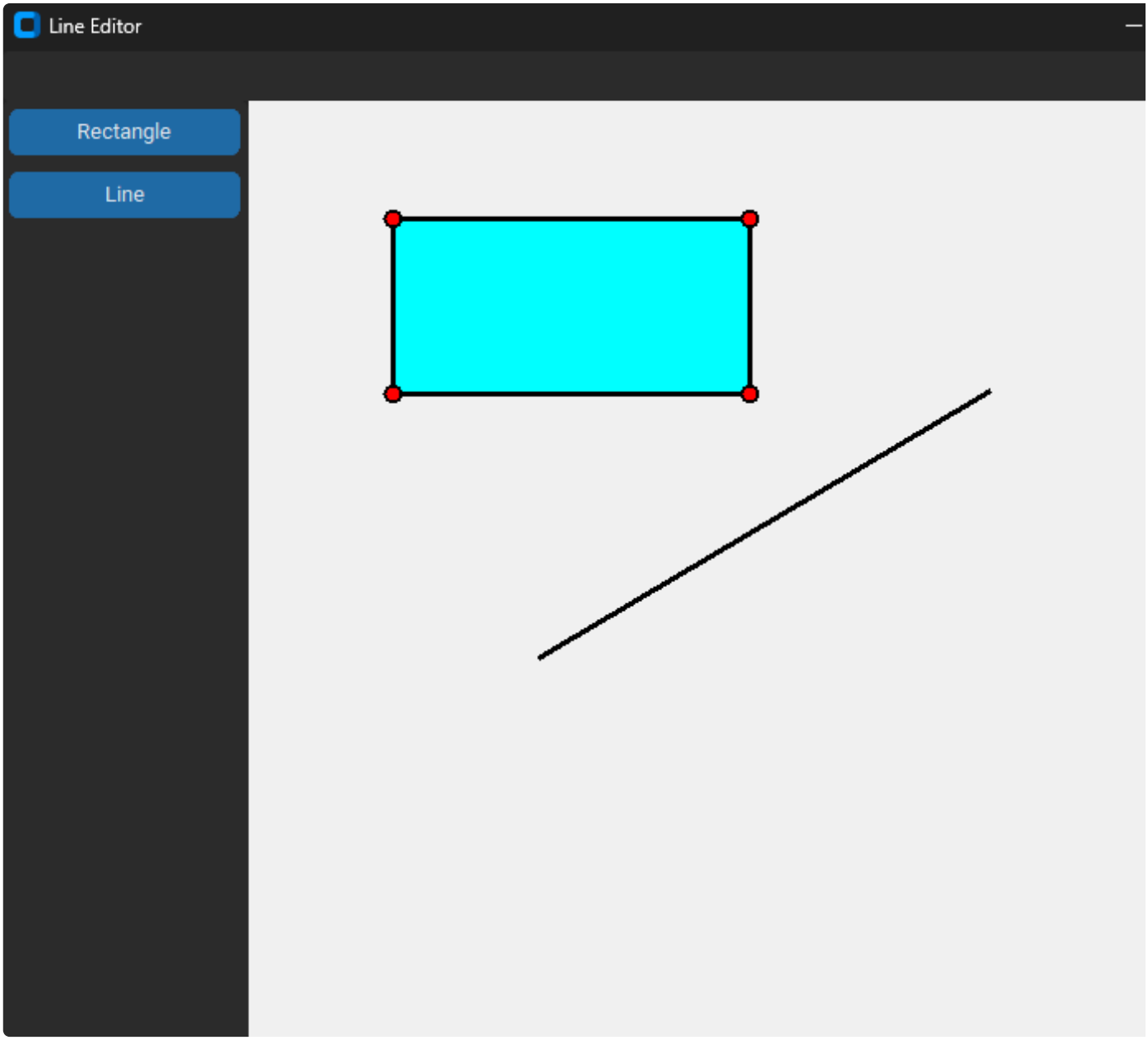
```
self.start.x, self.start.y
            self.current_shape_obj.x2, self.current_shape_obj.y2 = x, y
            self.canvas.draw_shapes()

    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.

```
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(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

            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')

    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 draw. This is how we ensure that the grid is drawn in the background behind all other shapes.

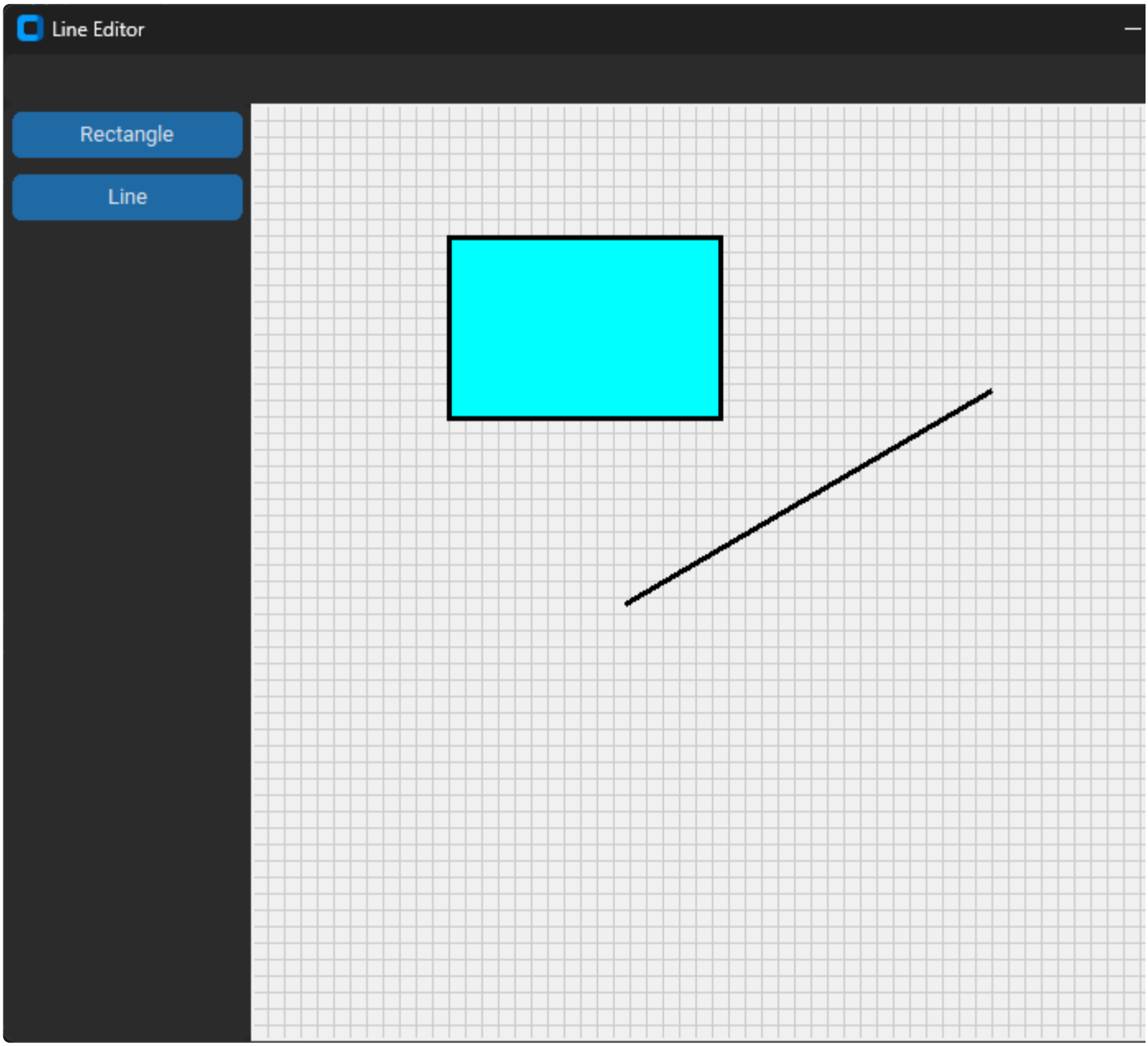
```
import customtkinter as ctk
from UI_Lib.mouse import Mouse
from Shape_Lib.line import Line
from Shape_Lib.grid import Grid


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 `snap_to_grid()` method whenever the line end coordinates are changed.

```python
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)
```

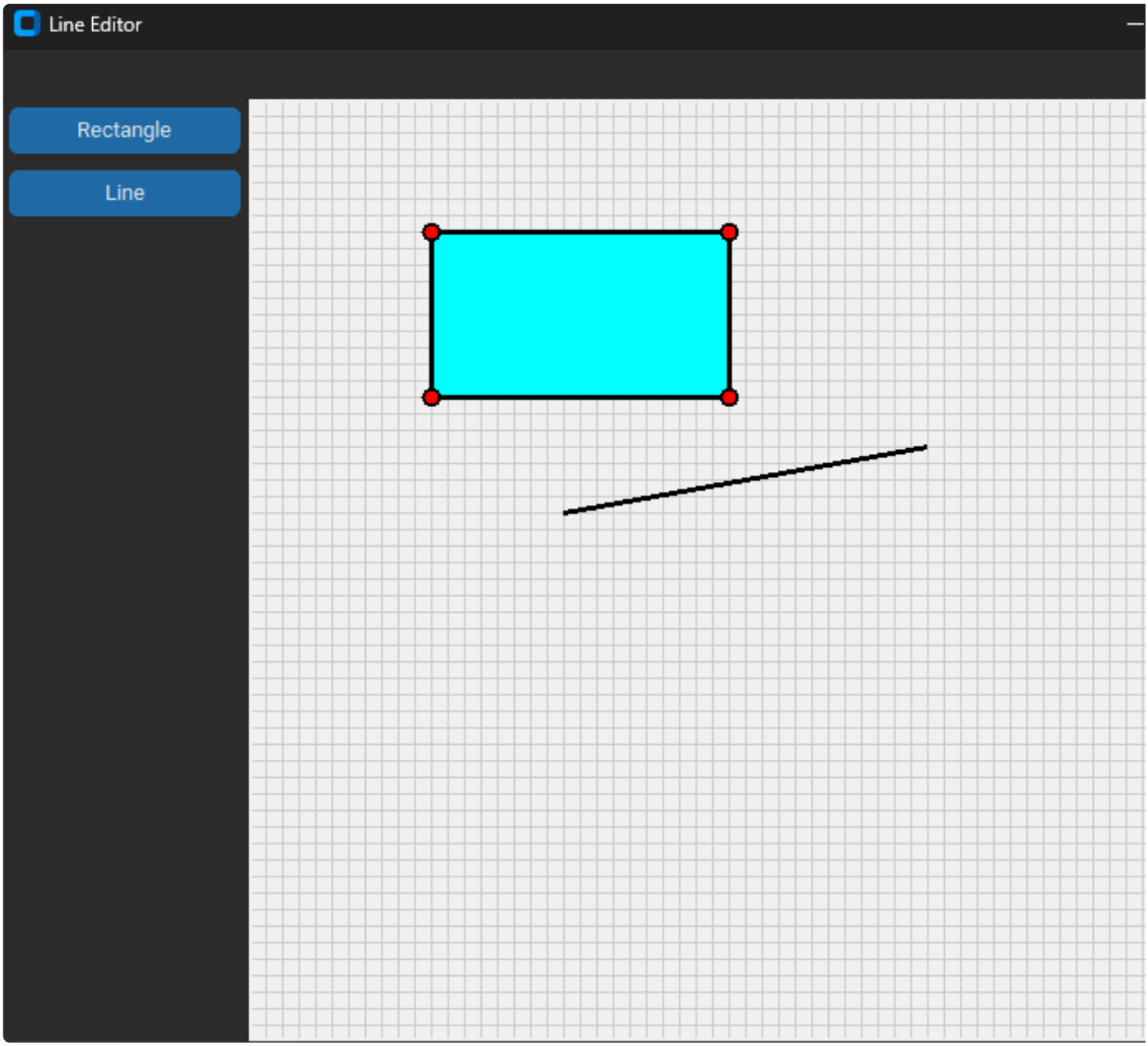
```
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    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
```

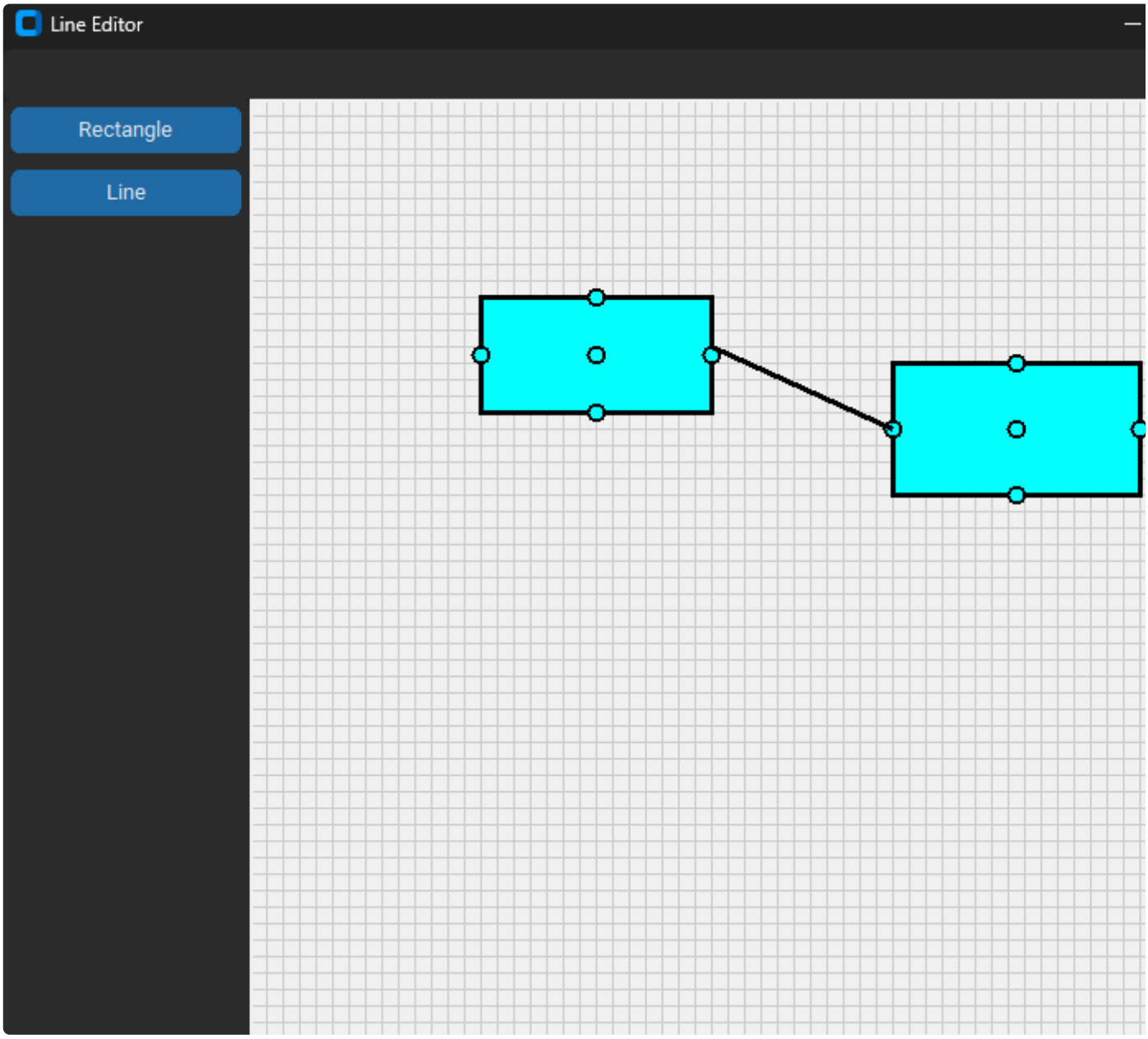
Modify the Left Frame Class to set the mouse mode to `"line_draw"` when the Line Button is selected.

```
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
```

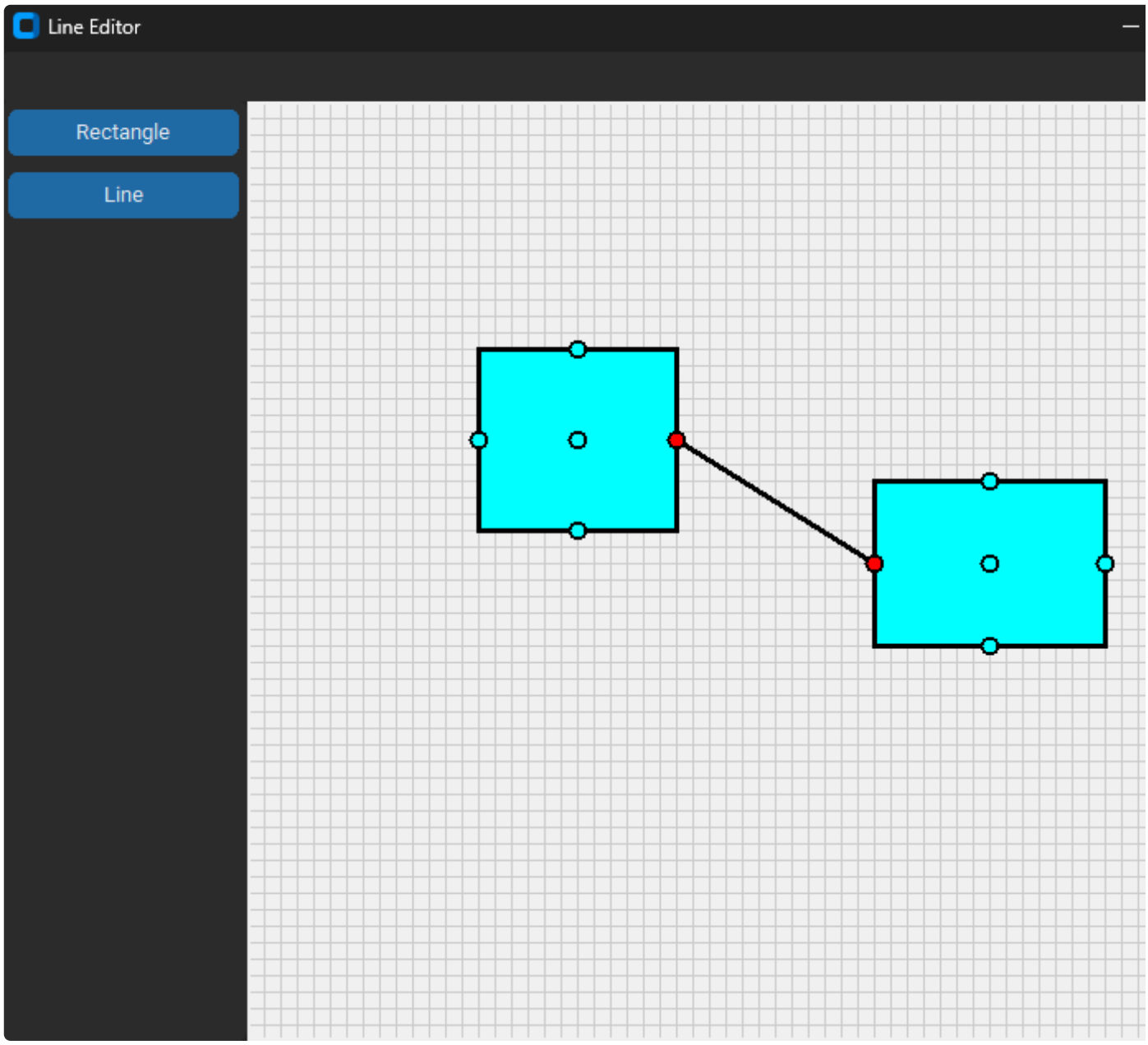
```
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.points = [self.x1, self.y1, self.x2, self.y2]
        self.is_selected = False

        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.

```
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 = []
        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):
        for conn in self.conn_list:
            if conn.conn_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
```

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)
```

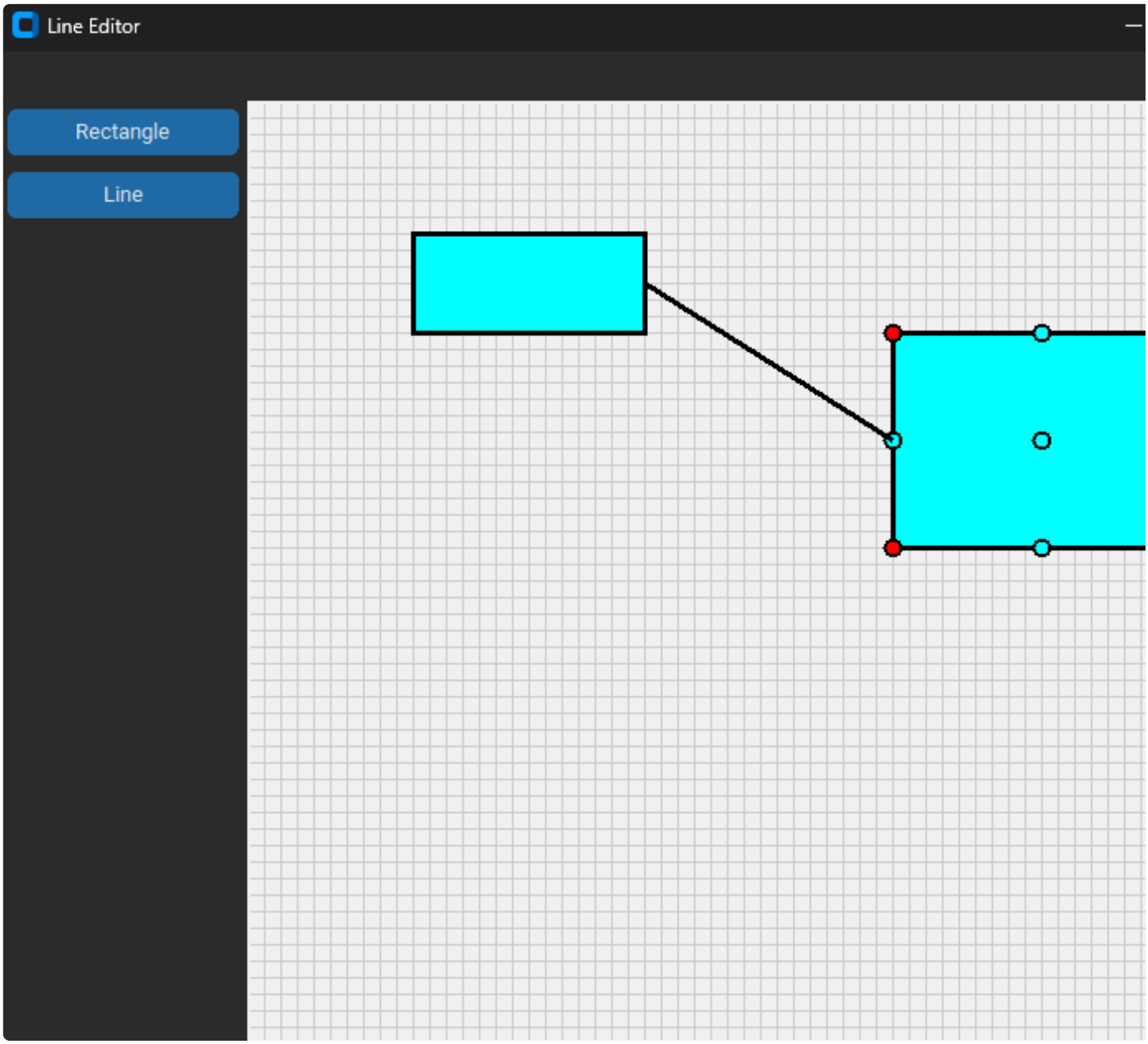
Modify the Rectangle Class to put the call to `move_connected_lines()` in the `draw_connectors()` method. Note that this eliminated calling it from move, resize, or rotate.

```
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.

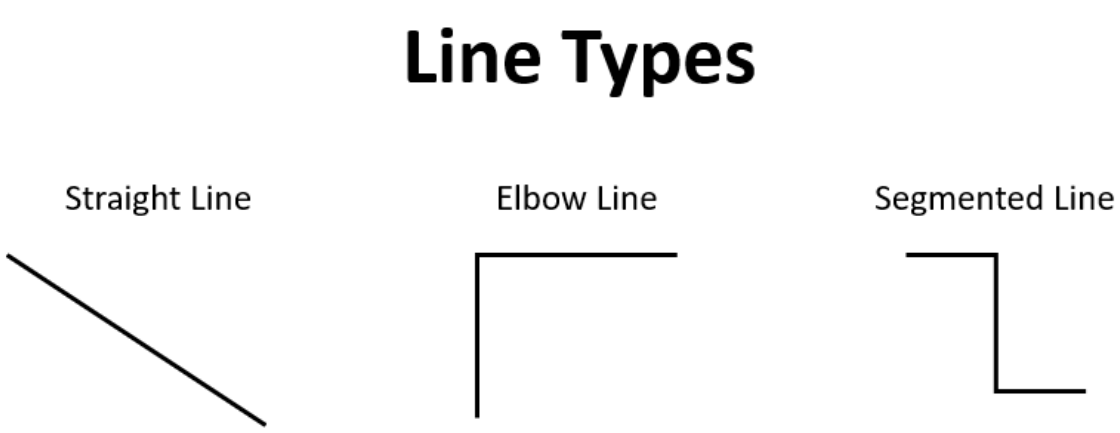


This took some refactoring to get it to work. The `__repr__` methods in the Connector and Connection classes were very useful to debug the code.

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.



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 draw 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()
```

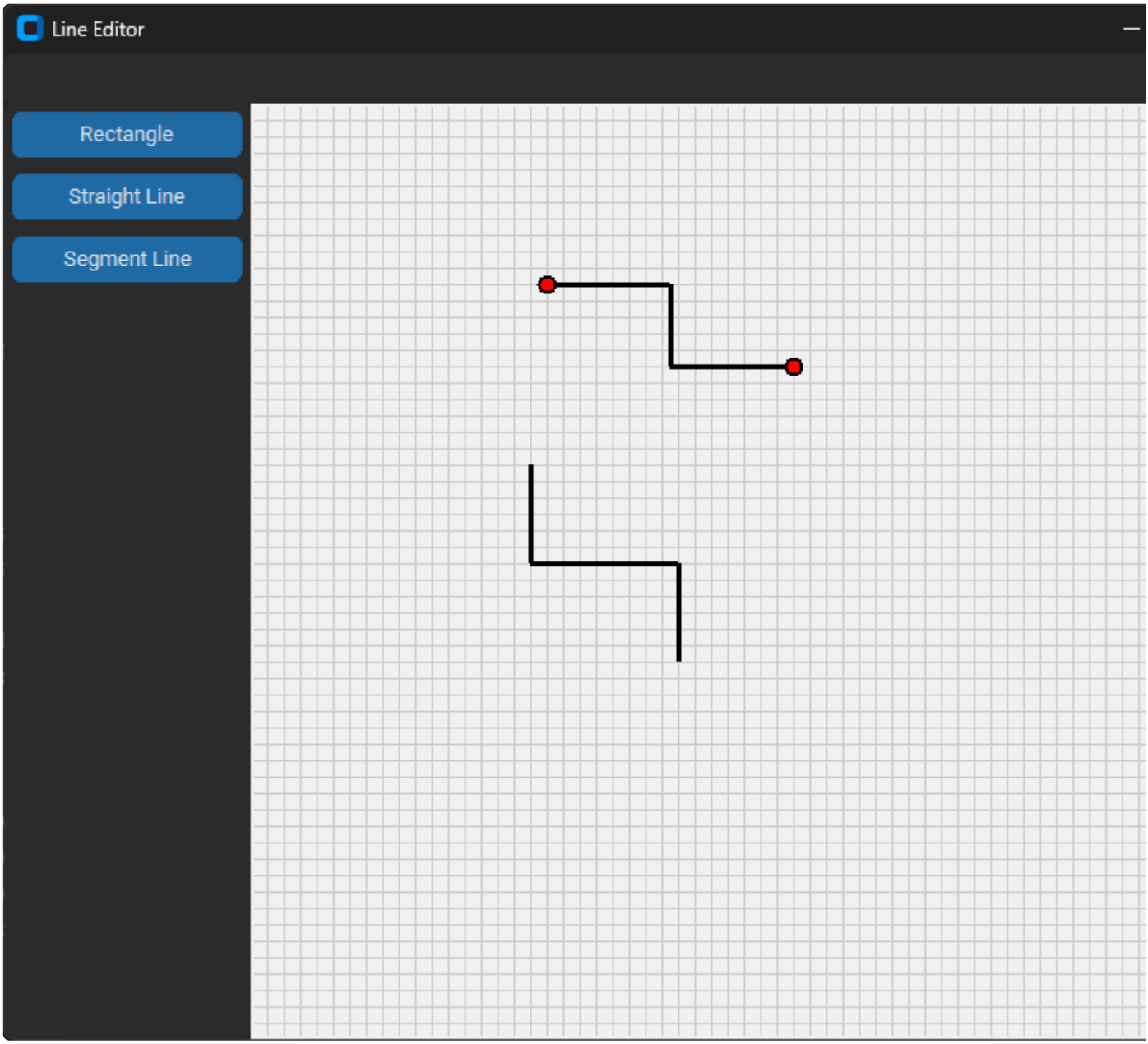
In the Mouse Class, add the code to draw a segment line to Draw left down method.

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()
```

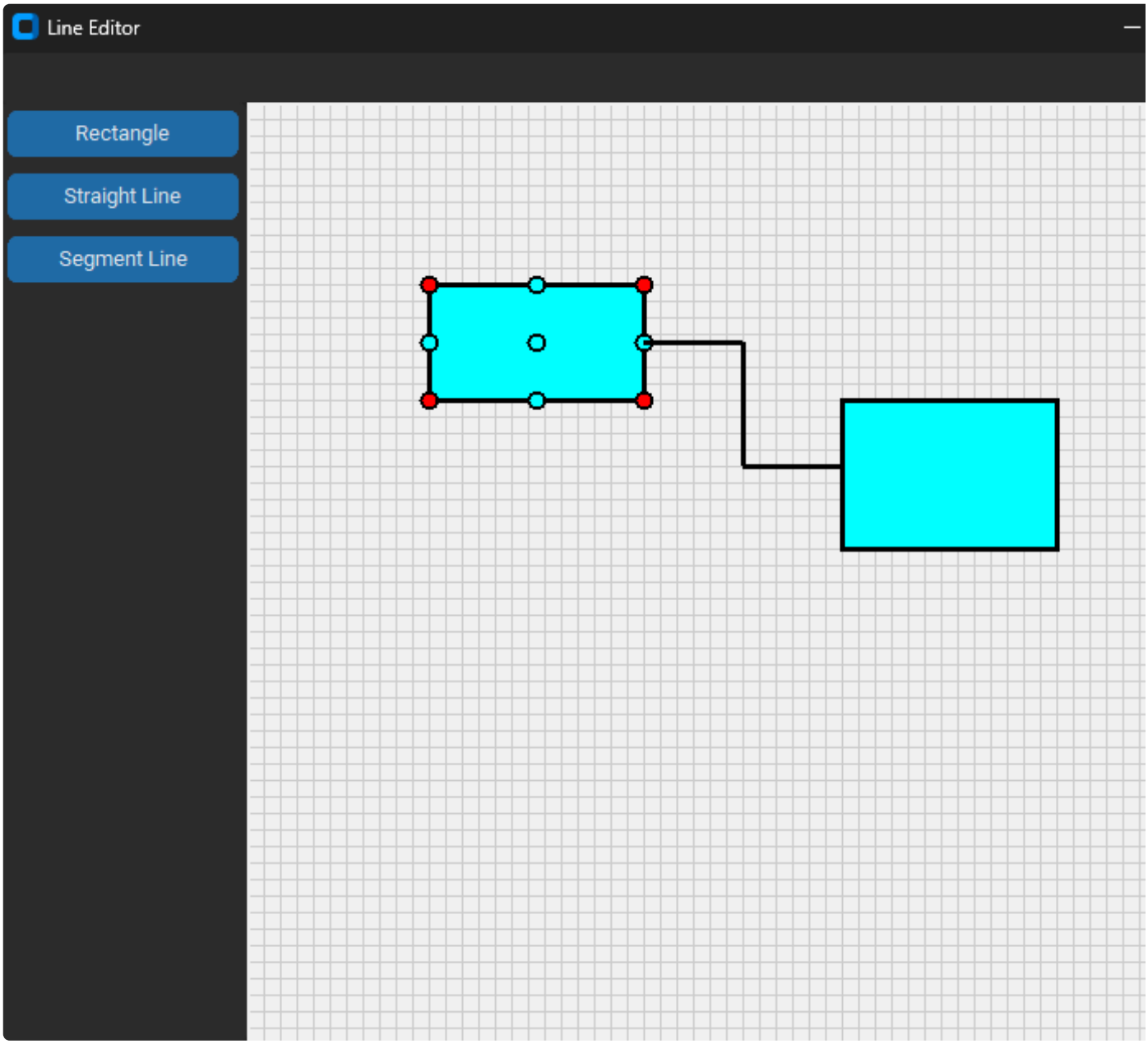
Mouse class lines of code: 177 lines

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



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.

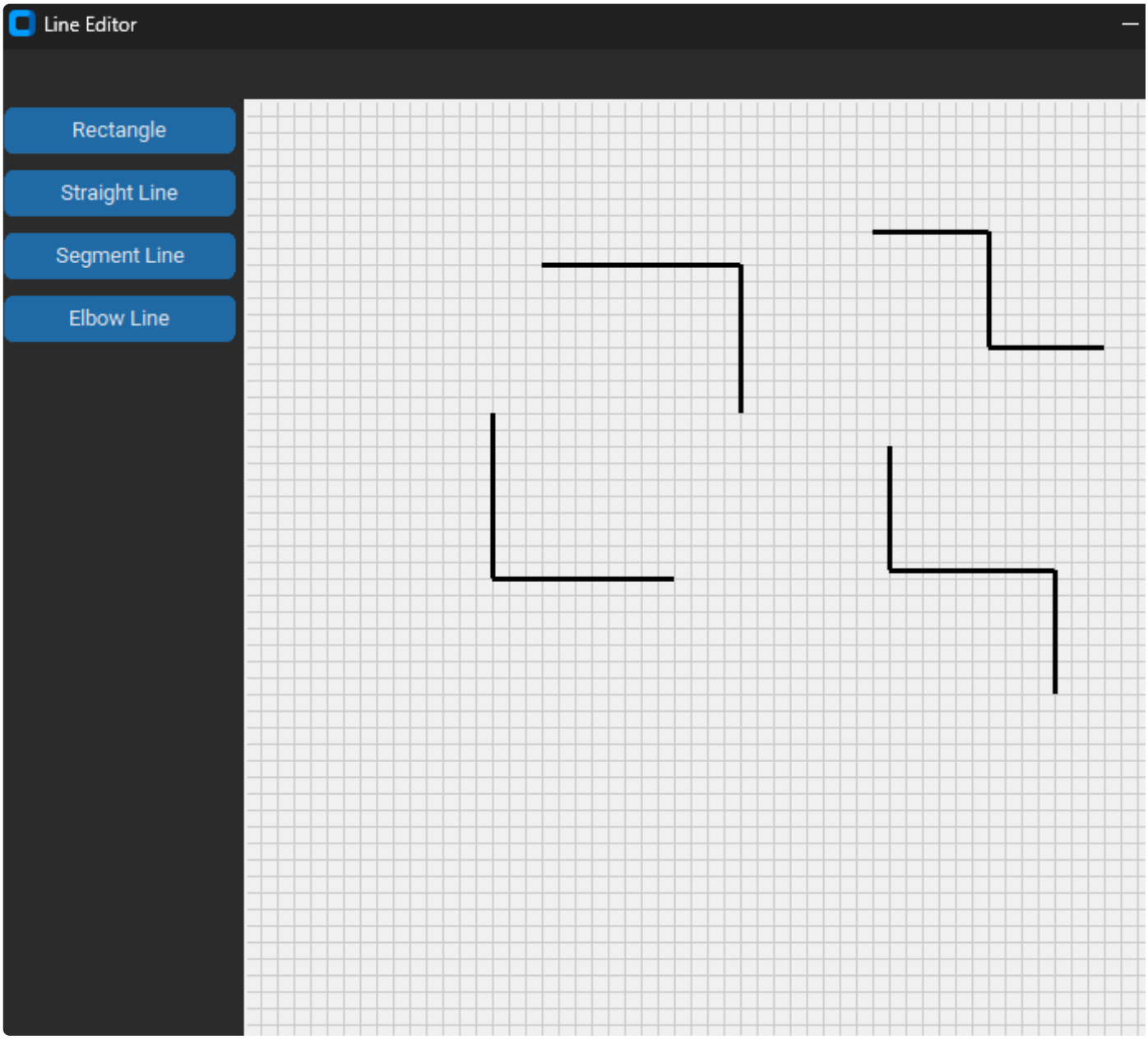
mouse.py

```
    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()
```

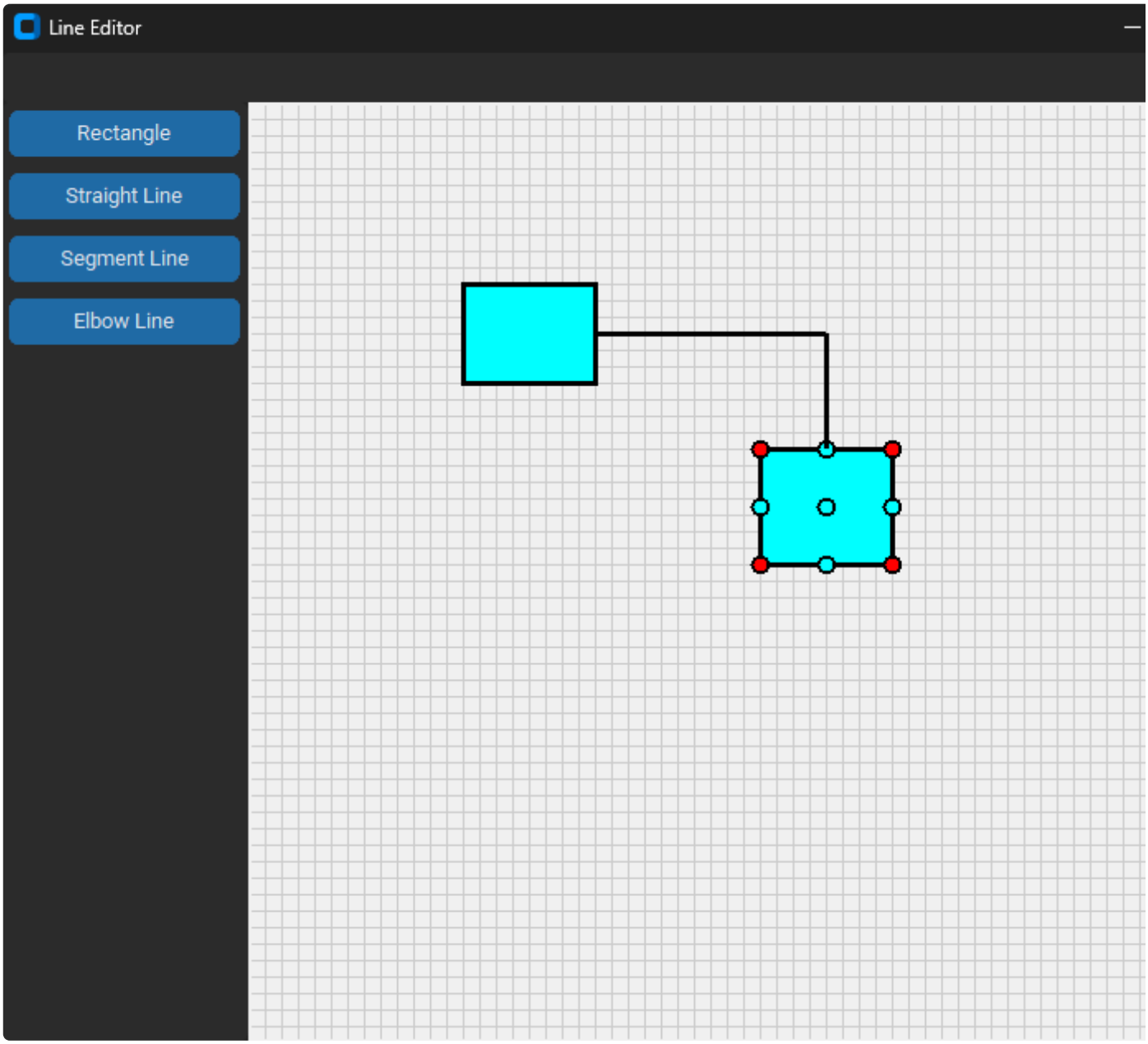
Mouse method lines of code: 182 lines

Run the program and draw elbow lines from the Left Menu. Try setting the starting direction using the 'h' and 'v' keys.



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, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)
```

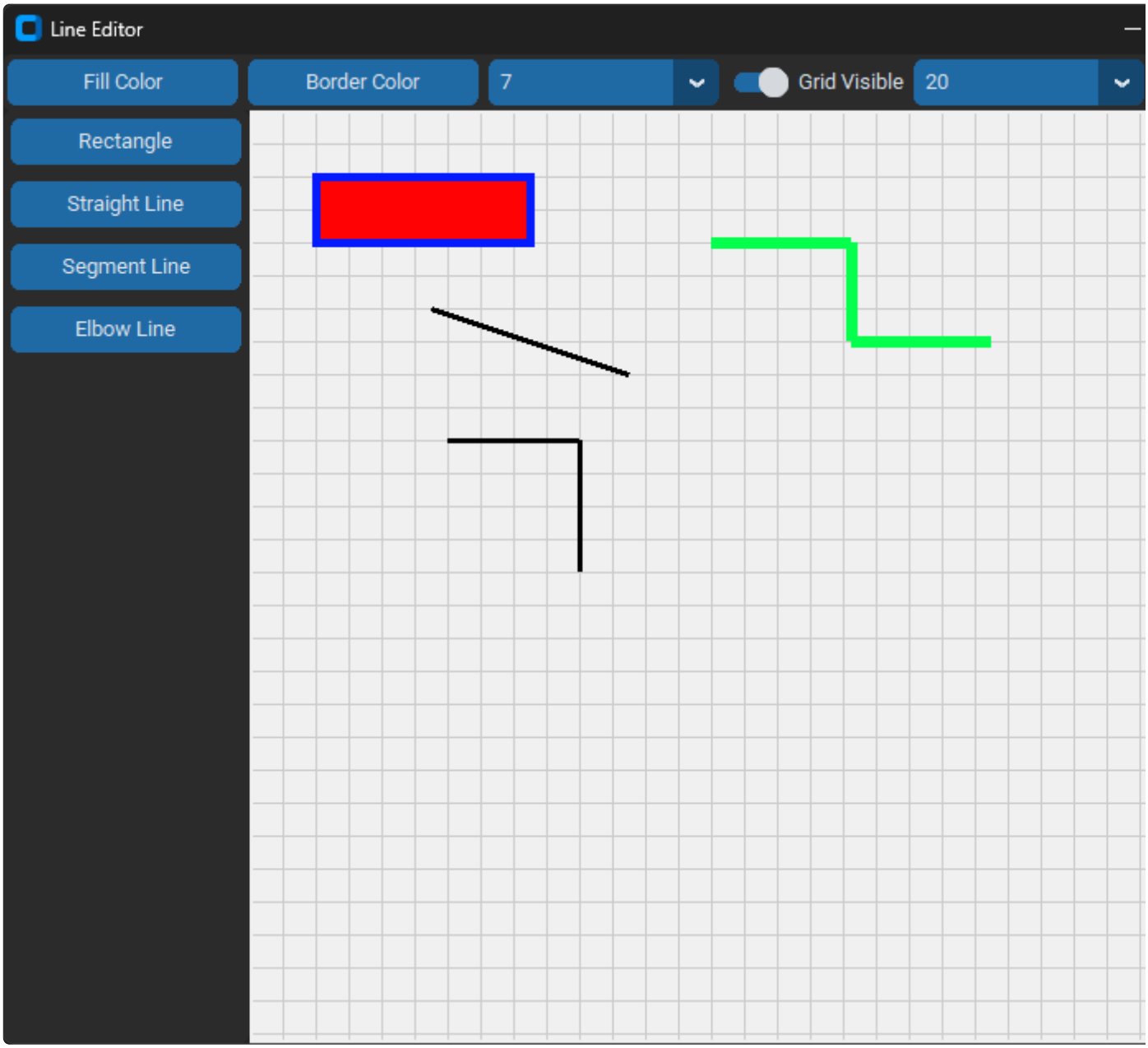
Modify the Elbow Line Class to set the fill color and border width.

```
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 .

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.