Chapter 5 - Diagram Editor

The first advanced project is a Diagram Editor which will integrate the techniques and solutions that were developed in the Shape Editor and Line Editor applications.

The Diagram Editor is a diagram creation application that allows the user to select shapes from a menu which creates the shape on the canvas. The user can draw, move, rotate, and resize the shape as well as change the fill color, border color, or border width.

Program Design & Architecture:

- Object-Oriented Programming (OOP) - class abstraction, encapsulation, inheritance, & polymorphism
- Modules - separate files from the main program file
- Package - directory with `__init__.py` file that imports all directory modules such that the program can use a standard import statement

Libraries

- Helper Library (Helper_Lib) - Utility classes or other modules that add helpful features
- Shape Library (Shape*Lib) - All shape classes that wrap the \`canvas.create*()`function where`\` is Rectangle, Oval, Polygon, Line, Text, Image, etc.
- User Interface Library (UI_Lib) - GUI interface classes for custom frames, canvas, keyboard, mouse, etc.

Shape Library

- Shape Class - base class for shape classes
- Rectangle Class
- Oval Class
- Triangle Class
- Straight Line Class
- Segment Line Class
- Elbow Line Class
- Text Class

- ✓ Picture Class
- ✓ Connector Class
- ✓ Selector Class
- ✓ Connection Class
- ✓ Grid Class
  - ✓ Snap to Grid
  - ✓ Grid visibility on/off
  - ✓ Snap size
  - ✓ Grid size

Shape Modification

- ✓ Keyboard rotate
- ✓ Set fill color
- ✓ Set border color
- ✓ Set border width
- ✓ Mouse draw
- ✓ Mouse move
- ✓ Mouse resize

User Interface Library

- ✓ Canvas Class
- ✓ Mouse Class
  - ✓ Mouse draw
  - ✓ Mouse move
  - ✓ Mouse resize
- ✓ Keyboard Class
  - ✓ Keyboard rotate
- ✓ Top Frame Class
  - ✓ File Menu Frame
  - ✓ Settings Frame
  - ✓ Help Frame
  - ✓ Shape Appearance Frame
    - ✓ Set fill color
    - ✓ Set border color
    - ✓ Set border width

- ✓ Left Frame Class
    - ✓ Shape Button Frame
    - ✓ Line Button Frame
- ✓ Bottom Frame Class
    - ✓ Message label

Helper Library

- ✓ Point Class

Image Library - not a python package

- ✓ Create images in Microsoft PowerPoint
- ✓ Adjust image dimension in Microsoft Paint 3D
- ✓ Use the `expand=True` setting in `image.rotate()` method call to avoid drawing a new image for each rotation angle
- ✓ Add attribution.txt file to give image authors credit

Icon Library - not a python package

- ✓ Download icons in 24x24 px format from flaticon.com
- ✓ Create buttons with icons or icons & text to enhance the GUI
- ✓ Add attribution.txt file to give icon authors credit

Project Setup

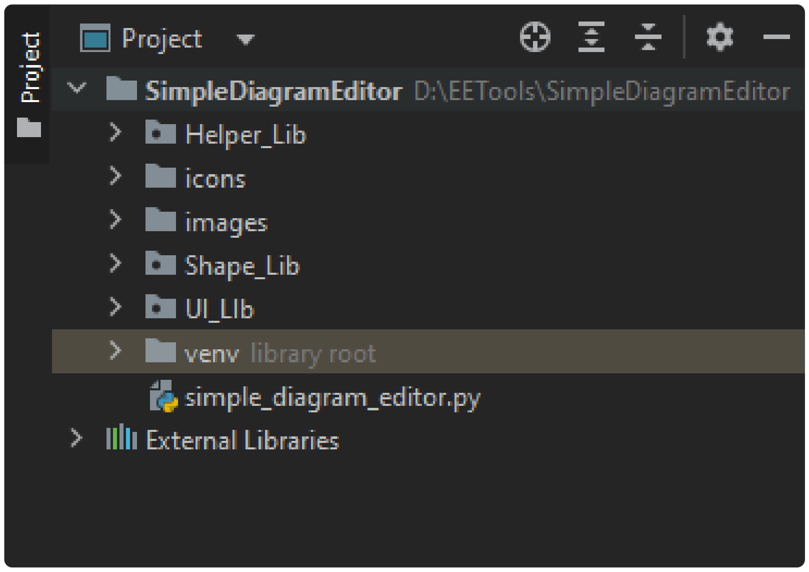
Language: Python 3.11
IDE: PyCharm 2023.2.1 (Community Edition)
Project directory: D:/EETools/SimpleDiagramEditor
Graphics library: CustomTkinter (https://customtkinter.tomschimansky.com/)

External libraries:

- ✓ pip install customtkinter
- ✓ python.exe -m pip install --upgrade pip
- ✓ pip install ctkcolorpicker

Initial Directory Structure

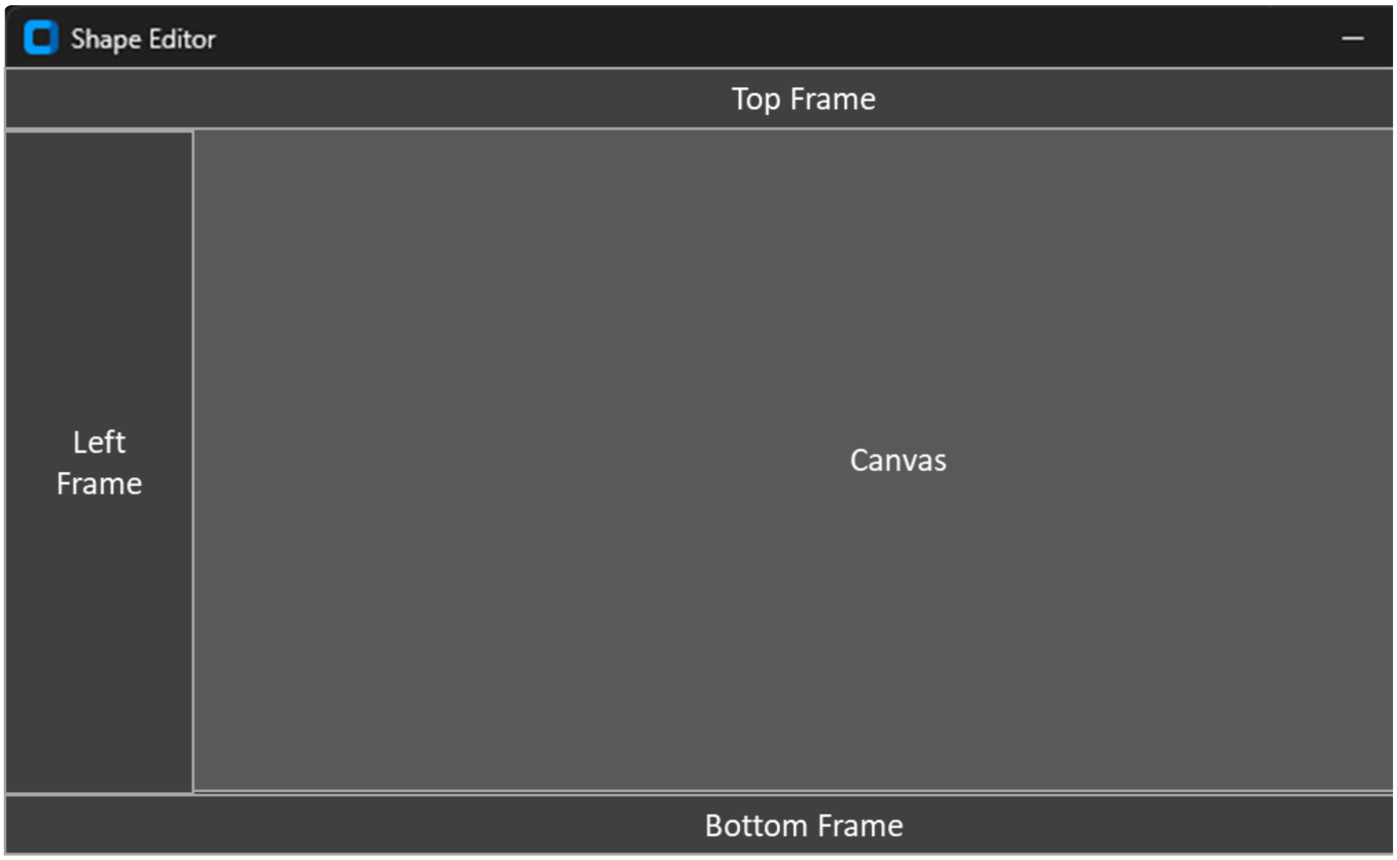
Per the specifications, the initial directory structure has three python packages: Helper_Lib, Shape_Lib, and UI_Lib. Recall that you must add `__init__.py` to each package directory to make it a package. We also need two standard directories: icons and images.



```
Project
SimpleDiagramEditor D:\EETools\SimpleDiagramEditor
    Helper_Lib
    icons
    images
    Shape_Lib
    UI_LIb
    venv library root
    simple_diagram_editor.py
External Libraries
```

User Interface - Simple Diagram Editor App Class

The initial user interface design is shown in the image below. It consists of a CustomTkinter window with a canvas widget and three custom frame widgets. Custom widgets allow us to create a modular UI design so that the main file class is small and manageable.



To understand this software development process, lets create the UI in the main file and then "modularize" the code by creating custom classes in module files.

Main Application Class

Create a new file in the Simple Diagram Editor called simple_diagram_editor.py.

Import statements are usually found at the top of the file. Initially, we import the CustomTkinter library which has widgets for creating a modern user interface with light and dark modes.

```
import customtkinter as ctk
```

Next we create the Simple Diagram Editor App Class which inherits from the ctk window class, so we will need two initializers: one for the derived class and one for the "super" or base class.

```
class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
```

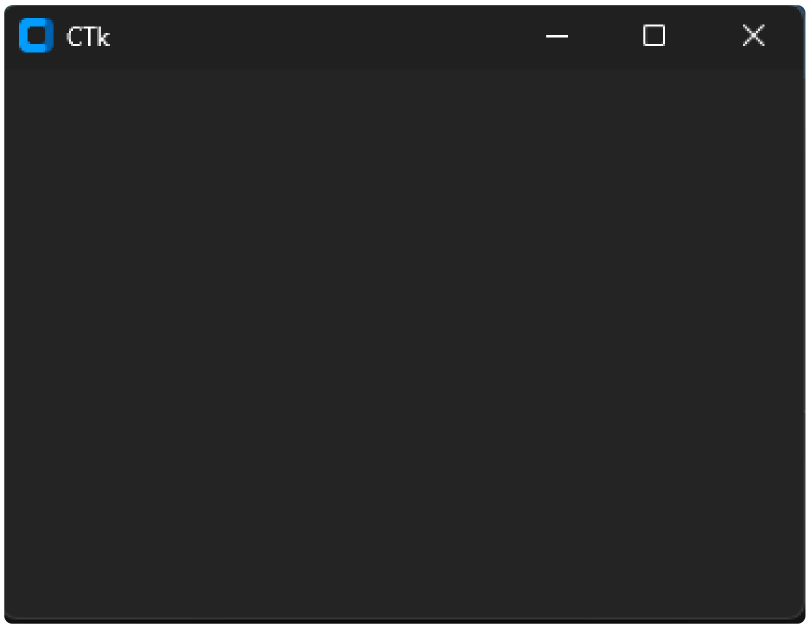
We can instantiate this class in the "main" entry point code as follows:

```
if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

Here is the complete simple_diagram_editor.py file at this point

```
import customtkinter as ctk


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")


if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

The program is now executable and if you run it, you will see a blank CustomTkinter window.

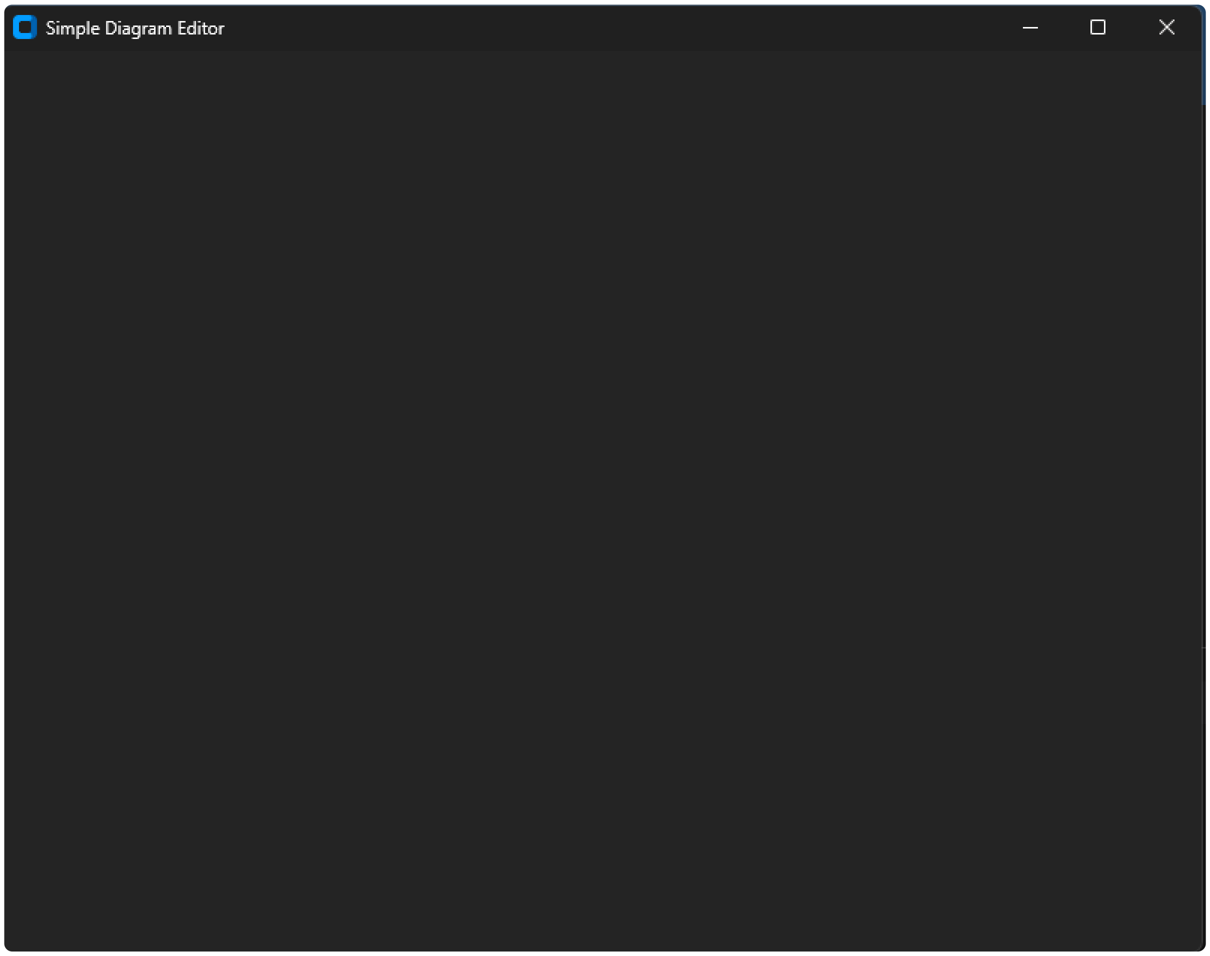


Not very exciting, but it proves that our project is setup and ready for development. Sort of a "Hello World" program with a user interface.

To improve the window, set the window geometry which defines its size and location using a weird string notation `"WxHxXxY"` where W = window width, H = window height, X = x location coordinate, and Y = y location coordinate in pixels. Set the window title to "Simple Diagram Editor".

```
. . .

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

. . .
```

A code snippet is indicated by the three dots (. . .). Note that I did not show the import statement or the main entry point code in this code snippet. You should assume that they are still there. Run the program again to see that the window size, location, and title are set.



Creating the User Interface

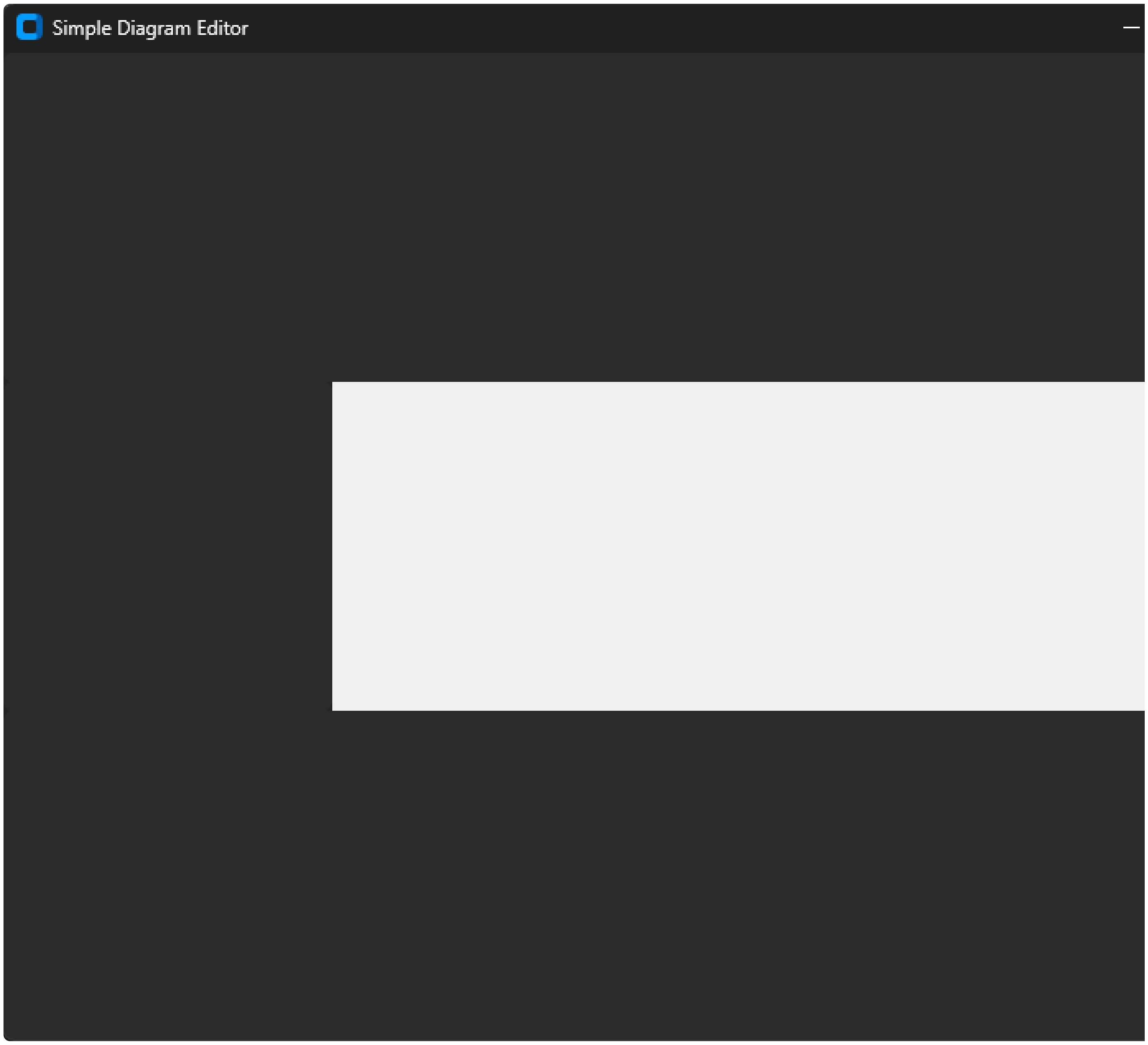
Add code that creates a canvas and three frames.

```
. . .

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = ctk.CTkCanvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

. . .
```

If you run the program now, there is no change. We need to add the new widgets to the window using one of the three tkinter layout managers: place, pack, or grid. For this application we will use the pack or packer layout manager which "packs" or creates a layout relative to earlier widgets so the order that widgets are packed is important in determining the relative position and size of the widget. We want the top and bottom frames to extend from the left side of the window to the right side so we pack those widgets first. Set the `side=` option to Top for the Top Frame and Bottom for the Bottom Frame. Then we add the left frame and the canvas which will be packed between the top and bottom frame. Set the `side=` option to Left for the Left Frame and Right for the Canvas.

```
. . .

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = ctk.CTkCanvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

. . .
```

Run the program to see the initial user interface "skeleton". It looks strange because the frames and canvas have "autosized" to fill the available space. Don't worry or attempt to adjust the frame sizes as we will take advantage of the autosize feature when widgets are added to the frame. The canvas will autosize also to take up all available space not used by the frames.



Now add test buttons to the Top Frame and Left Frame. Also, add a label to the Bottom Frame as follows.

```
. . .

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Add top frame widgets here
        button = ctk.CTkButton(self.top_frame, text="File")
        button.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add left frame widgets here
        button = ctk.CTkButton(self.left_frame, text="Shape")
        button.pack(side=ctk.TOP, padx=5, pady=5)

        # Add bottom frame widgets here
        label = ctk.CTkLabel(self.bottom_frame, text="Diagram Editor ready...")
        label.pack(side=ctk.LEFT, padx=5, pady=5)


if __name__ == "__main__":

. . .
```

Run the program now and note that the Top Frame, Left Frame, and Bottom Frame have autosized to the test widgets. Also, the canvas now fills the available space. If you maximize the window, note that all frames and the canvas resize automatically.

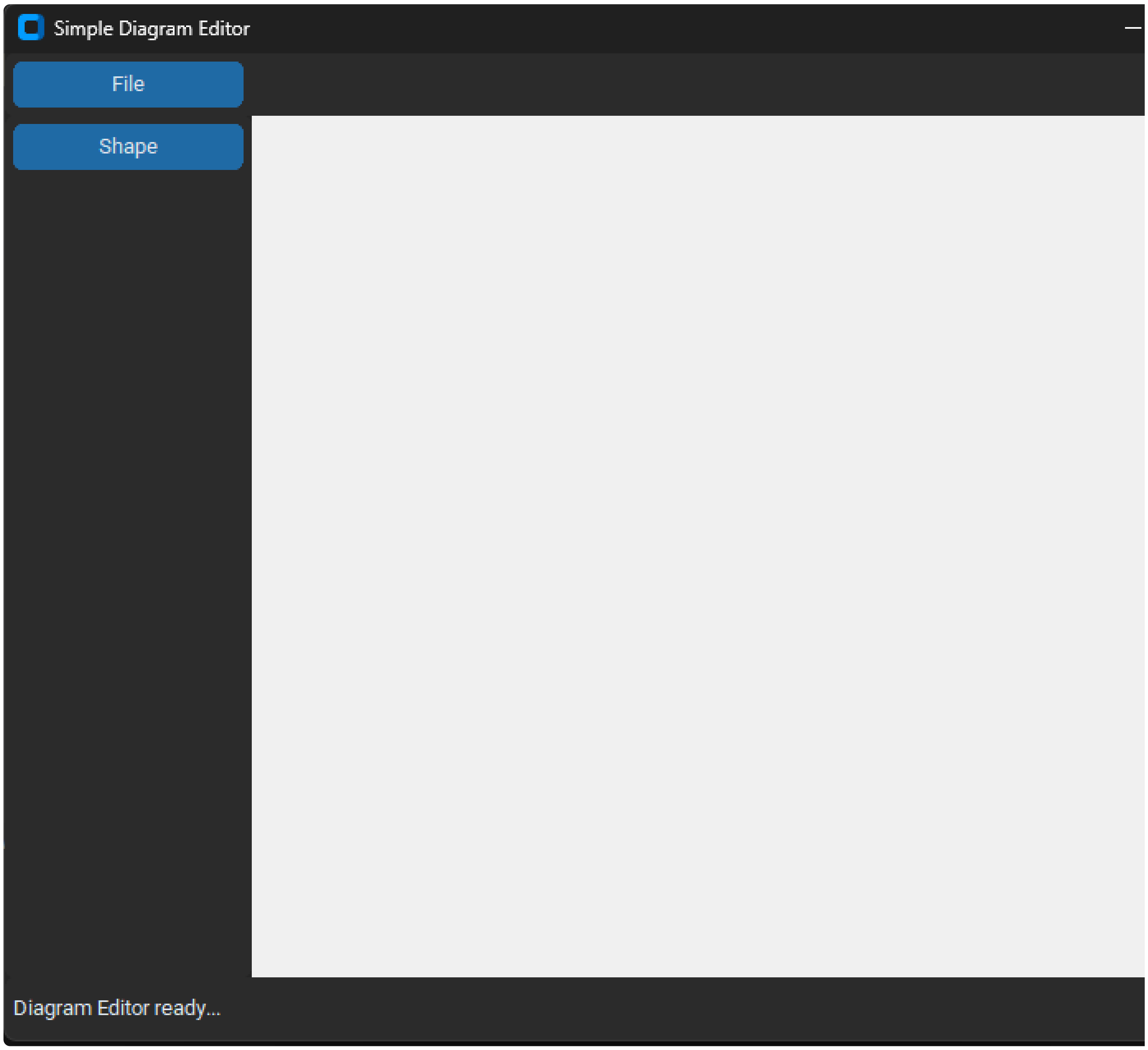


Diagram Editor ready...

Here is the full program code listing for simple_diagram_editor.py

```
import customtkinter as ctk


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

                # Create canvas and frames
        self.canvas = ctk.CTkCanvas(self)
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

                # Pack canvas and frames on the ctk window
        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Add top frame widgets here
        button = ctk.CTkButton(self.top_frame, text="File")
        button.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add left frame widgets here
        button = ctk.CTkButton(self.left_frame, text="Shape")
        button.pack(side=ctk.TOP, padx=5, pady=5)

        # Add bottom frame widgets here
        label = ctk.CTkLabel(self.bottom_frame, text="Diagram Editor ready...")
        label.pack(side=ctk.LEFT, padx=5, pady=5)


if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

We could keep adding widgets to the main diagram editor app but the code would quick expand and become more difficult to read and maintain. At this point, we will modularize the code into custom class modules.

Canvas Class

In the user interface library (UI_Lib), create a new file called canvas.py.

```
import customtkinter as ctk


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

Modify `UI_Lib/__init__.py` file to import the class

```
from UI_LIb.canvas import Canvas
```

The Canvas Class inherits from the CustomTkinter Canvas Class. To use this class we need to modify simple_diagram_editor.py to import the class and use it to instantiate our `self.canvas` object.

```
import customtkinter as ctk
from UI_LIb import Canvas  # Added new import


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)  # Changed to use the custom Canvas class
        self.top_frame = ctk.CTkFrame(self)
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)

. . .
```

If you run the program, you should see no change to the user interface.

Top Frame Class

The Top Frame class will contain widgets for File Menu, Settings Menu, Help Menu, and shape appearance controls such as shape fill color, shape border color, and shape border width.

In the user interface library (UI_Lib), create a new file called top_frame.py. The Top Frame class inherits from the CustomTkinter Frame Class. The Frame Class needs a "parent" widget where the frame will be created. Add the test button here for the Top Frame and remove it from the main diagram editor app. Note that the text button parent changes from `self.top_frame` to self which refers to the Top Frame Class.

```
import customtkinter as ctk


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


        # Add top frame widgets here
        button = ctk.CTkButton(self, text="File")  # Changed parent to self
        button.pack(side=ctk.LEFT, padx=5, pady=5)
```

Modify `UI_Lib/__init__.py` file to import the class

```
from UI_LIb.canvas import Canvas
from UI_LIb.top_frame import TopFrame
```

To use this class we need to modify simple_diagram_editor.py to import the class and use it to instantiate our `self.top_frame` object.

```
import customtkinter as ctk
from UI_LIb import Canvas, TopFrame  # Changed to import the TopFrame class


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)  # Changed to use the custom TopFrame
class
        self.left_frame = ctk.CTkFrame(self)
        self.bottom_frame = ctk.CTkFrame(self)
```

```
                # Deleted the top frame test button code

. . .
```

Running the program again, you will see that there is no change.

Left Frame Class

The Left Frame Class will contain the Shape creation buttons for creation of basic shapes such as Rectangles, Ovals, Triangles (Polygons), Lines, Text, and Images (Pictures).

Similar to the Top Frame Class, create a new module In the user interface library (UI_Lib) called left_frame.py. Add the test button here for the Left Frame and remove it from the main diagram editor app.

```
import customtkinter as ctk


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

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

Don't forget to modify `UI_Lib/__init__.py` file to import the class.

```
from UI_LIb.canvas import Canvas
from UI_LIb.top_frame import TopFrame
from UI_LIb.left_frame import LeftFrame
```

Modify the main file to import the Left Frame Class and use it to create the left frame object.

```
import customtkinter as ctk
from UI_LIb import Canvas, TopFrame, LeftFrame  # Added import for LeftFrame


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self)  # Changed to use the LeftFrame class
        self.bottom_frame = ctk.CTkFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

                # Deleted test buttons for TopFrame and LeftFrame

        # Add bottom frame widgets here
        label = ctk.CTkLabel(self.bottom_frame, text="Diagram Editor ready...")
        label.pack(side=ctk.LEFT, padx=5, pady=5)

. . .
```

Run the program and verify that there are no changes to the window.

Bottom Frame Class

The Bottom Frame Class will hold a label for user messages, modes, and status.

Create the Bottom Frame Class module called bottom_frame.py in the UI_Lib directory. Don't forget to add the test label.

```
import customtkinter as ctk


class BottomFrame(ctk.CTkFrame):
    def __init__(self, parent):
        super().__init__(parent)

        self.message = ctk.CTkLabel(self, text="Diagram editor ready...")
        self.message.pack(side=ctk.LEFT, padx=5, pady=5)
```

`UI_LIb/__init__.py`

```
from UI_LIb.canvas import Canvas
from UI_LIb.top_frame import TopFrame
from UI_LIb.left_frame import LeftFrame
from UI_LIb.bottom_frame import BottomFrame  # Added import for ButtonFrame
```

Update main file

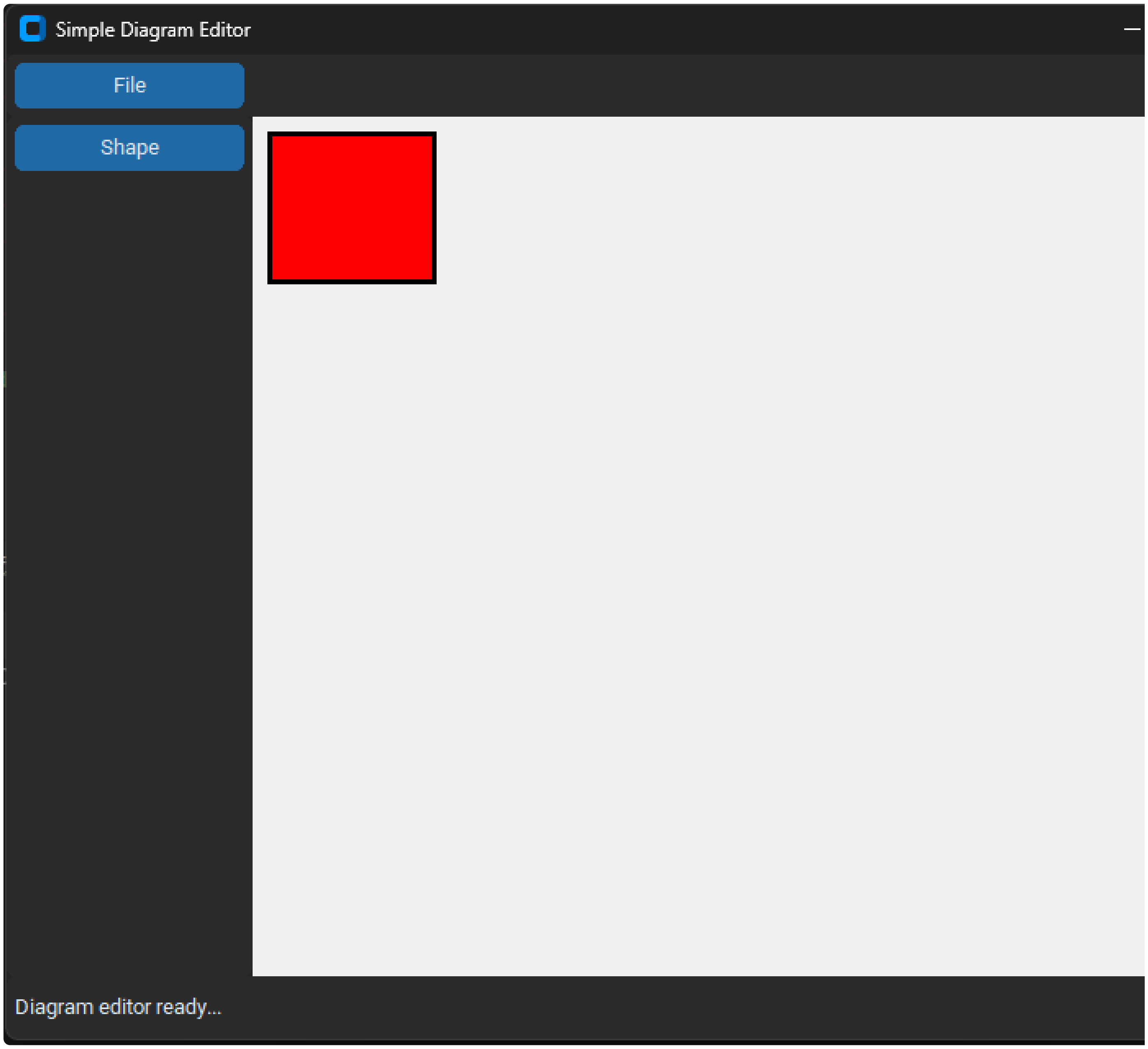
```
import customtkinter as ctk
from UI_LIb import Canvas, TopFrame, LeftFrame, BottomFrame  # Added import for
BottomFrame


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self)
        self.bottom_frame = BottomFrame(self)  # Changed to use the BottomFrame
class

                # Delected code for test buttons and test label

. . .
```

Again, run the program to verify that there are no changes to the simple user interface.

Rectangle Class

We could create a simple rectangle from the main application in one line of code as follows:
simple_diagram_editor.py

```
. . .

class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self)
        self.bottom_frame = BottomFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Add shapes here
        rect = self.canvas.create_rectangle(10, 10, 110, 100, fill="red",
outline="black", width=3)  # Create a rectangle!

. . .
```



Cool! We created a Rectangle with fill color = "red", border (outline) color = "black", and border width = 3. We positioned the shape at the point 10, 10 which is the upper-left corner of the shape and at point 110, 110 which is the lower-right corner. Note that the width and height are 100 px.

The Canvas has shape creation methods for many shapes:

- Rectangle (`canvas.create_rectange(x1, y1, x2, y2, fill="red", outline="black", width=3)`)
- Oval (`canvas.create_oval(x1, y1, x2, y2, fill="red", outline="black", width=3)`)
- Polygon (`canvas.create_polygon(points, fill="red", outline="black", width=3)`) where points is a list of points

- Arc (`canvas.create_arc(x1, y1, x2, y2, fill="red", outline="black", width=3)`)
- Bitmap (`canvas.create_bitmap(x, y, bitmap=filename, anchor=ctk.CENTER)`)
- Image (`canvas.create_image(x, y, image=filename, anchor=ctk.CENTER)`)
- Line (`canvas.create_line(x1, y1, x2, y2, fill="red", width=3)`)
- Text (`canvas.create_text(x, y, text="", anchor=ctk.CENTER)`)
- Window(`canvas.create_window(x, y, window=w, anchor=ctk.CENTER)`)

Canvas also has methods for modifying a shape drawn on it. Here are some methods we will use:

- `canvas.bbox(tagOrId)` - Returns a tuple (x1, y1, x2, y2) describing a rectangle that encloses all the objects specified by `tagOrId`. We will need this to get the dimensions and location for Text and Image shapes.
- `canvas.coords(tagOrId, points)` - Use to move a shape by passing a new set of points or coordinates.
- `canvas.delete(tagOrId)` - Deletes a shape from the canvas
- `canvas.itemconfigure(tagOrId, option)` - Use to change a shape color or width.

Rectangle Class

The Rectangle Class wraps the `canvas.create_rectangle()` method in a class structure so we can take control of the rectangle shape in an object-oriented way.

In the Shape Library (Shape_Lib) create a new file called rectangle.py. Name the class Rectangle. In the class initializer we need a reference to the canvas for the `self.canvas.create_rectangle()` method. Create attributes for the rectangle coordinates and set default values for fill color, border color, and border width.

rectangle.py

```
class Rect:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3
```

Next, add a place to store the rectangle canvas id which is an integer index to the canvas shape list. Add a create() method that creates the rectangle and sets the shape id.

```
class Rect:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None

    def create(self):
        """Create the shape once!"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)
```

In `Shape_Lib/__init__.py` import the Rectangle Class

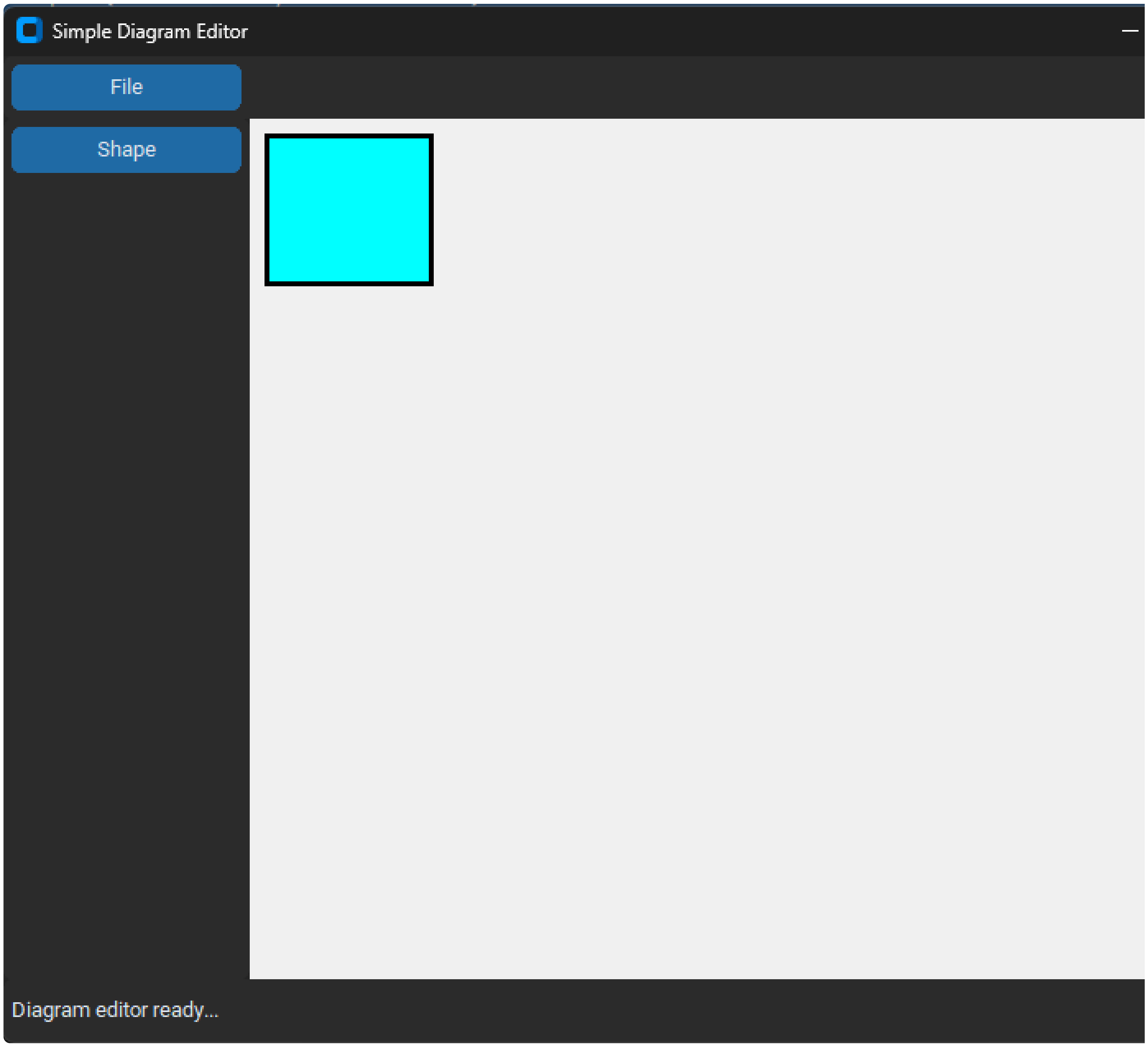
```
from Shape_Lib.rectangle import Rectangle
```

This is enough code to create a rectangle. In the main application file, import the Rectangle class and replace the code that creates a rectangle with code that instantiates a rectangle object and calls the object create() method.

```
import customtkinter as ctk
from UI_LIb import Canvas, TopFrame, LeftFrame, BottomFrame
from Shape_Lib import Rectangle  # Add import for Rectangle
```

```
class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self)
        self.left_frame = LeftFrame(self)
        self.bottom_frame = BottomFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        # Add shapes here
        # rect = self.canvas.create_rectangle(10, 10, 110, 100, fill="red",
outline="black", width=3)  # Commented out old code
        rect = Rectangle(self.canvas, 10, 10, 110, 100)  # Added code to use
the Rectangle Class
        rect.create() # Call the Rectangle create() method
```

We now have a cyan rectangle drawn from the Rectangle Class.



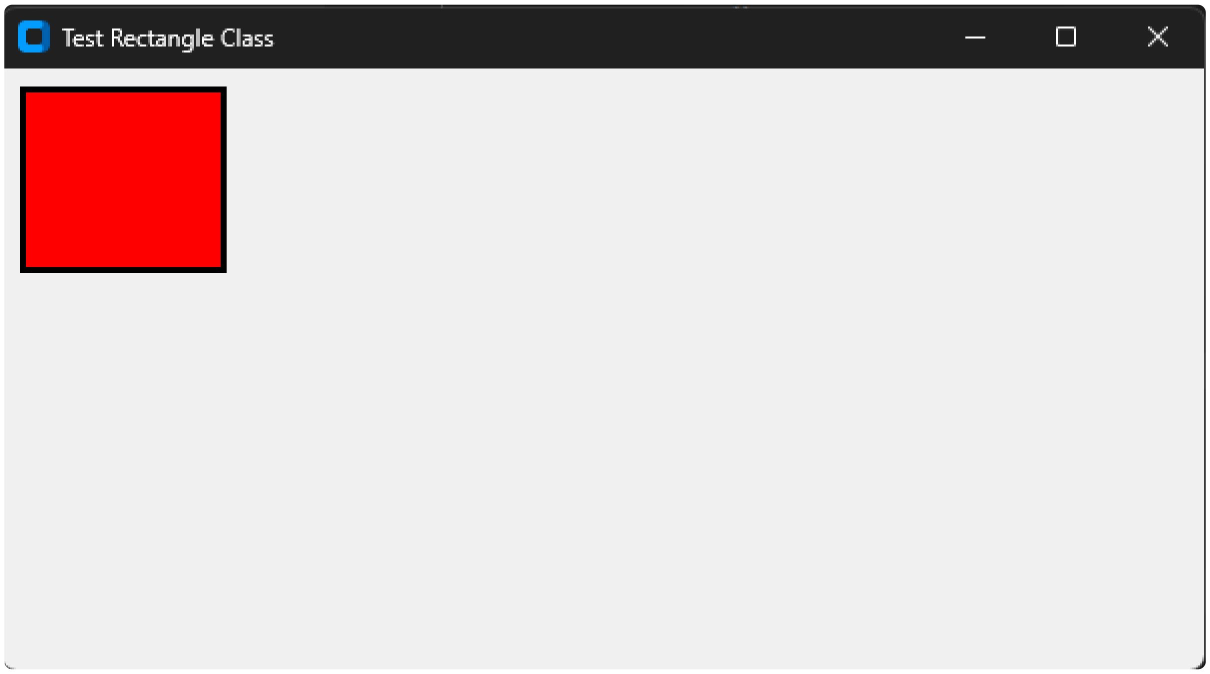
Next add an update() method to the Rectangle Class

```
class Rectangle:
    def __init__(self, canvas, x1, y1, x2, y2):
        self.canvas = canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3
        self.id = None

    def create(self):
        """Create the shape once!"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)

    def update(self):
        """Update the shape, don't recreate it!"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)
```

We can test our classes locally before importing them into the main application. Lets add a `__repr__()` method that creates a custom string for the print() methods. Also, we can add a main section to test the class.

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

        self.id = None

    def create(self):
        """Create the shape once!"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)
    def update(self):
        """Update the shape, don't recreate it!"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

    def __repr__(self):
        return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))


if __name__ == "__main__":
    import customtkinter as ctk

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Rectangle Class")

    canvas = ctk.CTkCanvas()
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    rect = Rectangle(canvas, 10, 10, 110, 100)
    rect.create()
    print(rect)

    # Change the fill color to test the update() method
    rect.fill_color = "red"
    rect.update()

    app.mainloop()
```

Run the Rectangle Class to see the test window.



Console Output

```
Rectangle: x1, y1, x2, y2 = (10, 10, 100, 100)
fill color: cyan
border_color: black
border_width: 3
```

Ok, all Rectangle Class tests passed so lets add the class to the main application.

Now lets create the rectangle from the Left Frame shape menu, we can modify the test button to perform this task.

In UI_Lib/left_frame.py, import the Rectangle Class

```
import customtkinter as ctk
from Shape_Lib.rectangle import Rectangle
```

In the class initializer, we will need a reference to the application canvas

```
class LeftFrame(ctk.CTkFrame):
    def __init__(self, parent, canvas):  # Added canvas to the initializer
```

```
        super().__init__(parent)
        self.canvas = canvas  # Create a local canvas attribute

        # Add left frame widgets here
        rect_button = ctk.CTkButton(self, text="Rectangle",
command=self.create_rect)  # Change to Rectangle button with command
        rect_button.pack(side=ctk.TOP, padx=5, pady=5) # Change to Rectangle
button
```

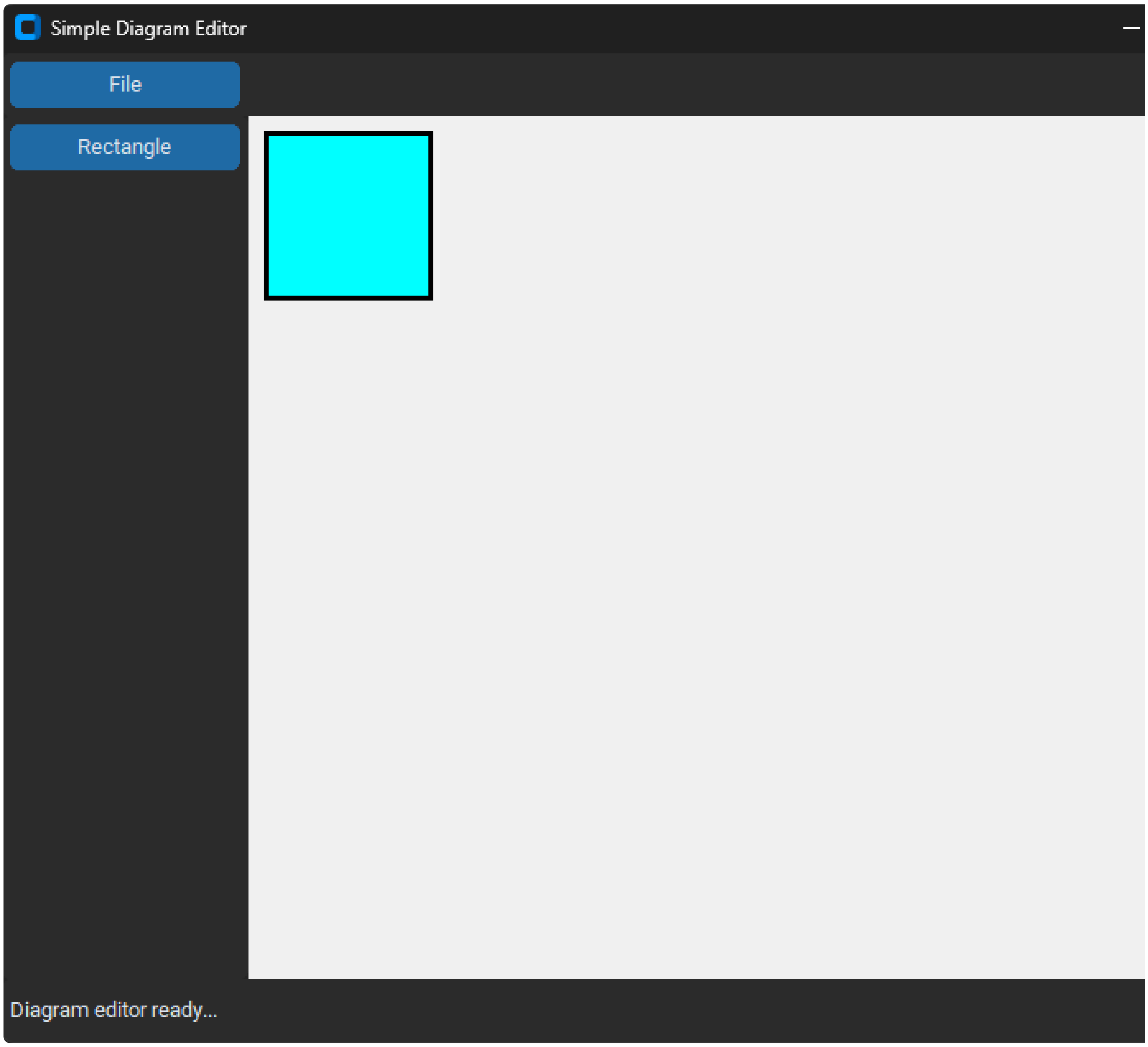
Add the `create_rect()` method to handle creation of the rectangle

```
    def create_rect(self):
        shape = Rectangle(self.canvas, 10, 10, 110, 110)
        shape.create()
```

Here is the complete Left Frame class definition

```
import customtkinter as ctk
from Shape_Lib.rectangle import Rectangle


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

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

    # Left frame button handlers
    def create_rect(self):
        shape = Rectangle(self.canvas, 10, 10, 110, 110)
        shape.create()
        self.canvas.shape_list.append(shape)
        self.canvas.mouse.selected_shape = shape
```

Run the program, click the Rectangle button and confirm that a cyan rectangle appears on the canvas.



Now that the rectangle shape can be drawn on the canvas from the Left Frame menu, lets add the following capabilities:

- Mouse move
- Keyboard rotate
- Set fill color
- Set border color
- Set border width
- Mouse draw
- Mouse resize

Point Class

Lets enhance the Point Class created in the Line Editor Application to add a `__repr__()` method and a main test block to test the class.

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

    def __repr__(self):
        return "Point: (" + str(self.x) + ", " + str(self.y) + ")"


if __name__ == "__main__":
    a_point = Point(10, 15)
    print(a_point)
    print("a_point.x = ", a_point.x)
    print("a_point.y = ", a_point.y)
```

Console Output after running the program.

```
Point: (10, 15)
a_point.x =  10
a_point.y =  15
```

That was pretty easy and we can use this for debug during development.

Move Shape with Mouse

All mouse events will be handled by the Mouse Class.

- Import the Point Class
- Initialize the Mouse Class with a reference to the canvas
- Add an attribute to keep track of the currently selected shape
- Create two offset points and initialize
- Create bindings for left mouse down, drag, and up

- Create event handlers for left mouse down, draw, and up to move the selected shape
- Add a hit test to determine if a shape is selected by left mouse down

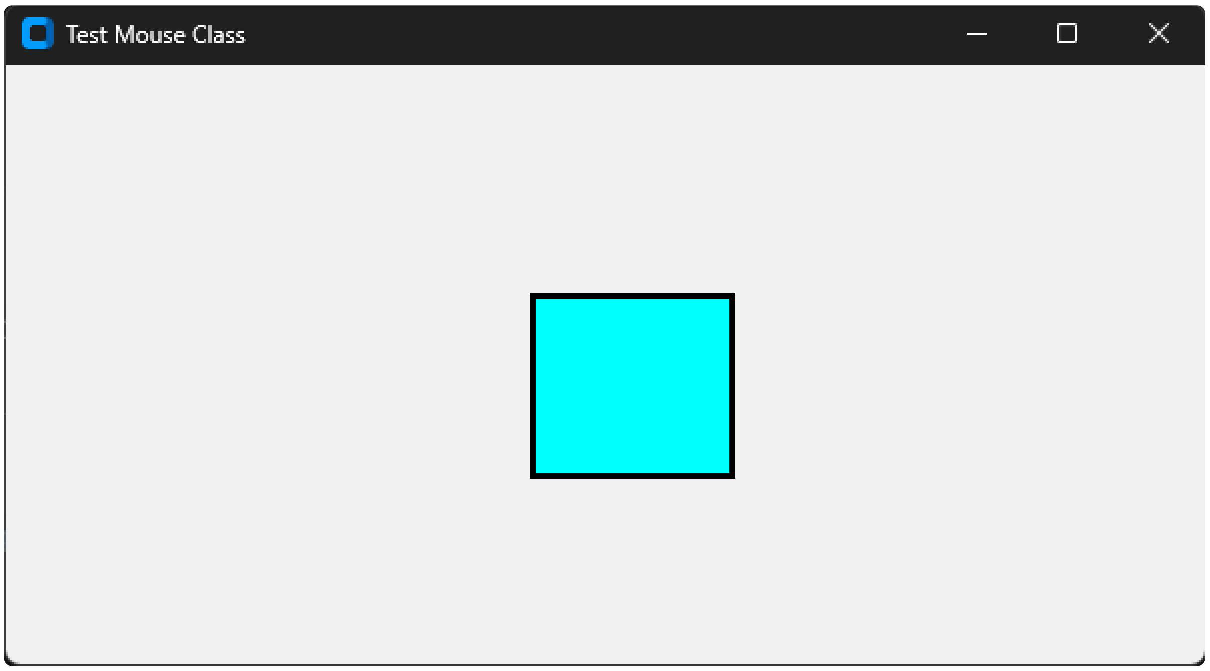
mouse.py

```python
from Helper_Lib.point import Point


class Mouse:
    def __init__(self, canvas):
        """Class to manage mouse events"""
        self.canvas = canvas
        self.selected_shape = None

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

        # Mouse bindings
        self.canvas.bind("<Button-1>", self.move_left_down)
        self.canvas.bind("<B1-Motion>", self.move_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.move_left_up)

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)
        if self.selected_shape:
            # print("Shape found: ", self.selected_shape)
            x1, y1 = self.selected_shape.x1, self.selected_shape.y1
            x2, y2 = self.selected_shape.x2, self.selected_shape.y2
            self.offset1.x = x - x1
            self.offset1.y = y - y1
            self.offset2.x = x - x2
            self.offset2.y = y - y2

    def move_left_drag(self, event):
        if self.selected_shape:
            x = event.x - self.offset1.x
            y = event.y - self.offset1.y
            self.selected_shape.x1, self.selected_shape.y1 = x, y
            x = event.x - self.offset2.x
            y = event.y - self.offset2.y
            self.selected_shape.x2, self.selected_shape.y2 = x, y
            self.canvas.redraw_shapes()
    def move_left_up(self, _event):
        self.offset1.x, self.offset1.y = 0, 0
        self.offset2.x, self.offset2.y = 0, 0

    def select_hit_test(self, x, y):
        for s in self.canvas.shape_list:
            if s.x1 <= x <= s.x2 and s.y1 <= y <= s.y2:
                self.selected_shape = s
```

We can add a main test section to test the mouse before it is integrated into the main application.

```
. . .

if __name__ == "__main__":
    import customtkinter as ctk
    from Shape_Lib import Rectangle
    from UI_LIb import Mouse, Canvas

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Mouse Class")

    canvas = Canvas(app)
    mouse = Mouse(canvas)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    rect = Rectangle(canvas, 10, 10, 110, 100)
    rect.create()
    canvas.shape_list.append(rect)

    app.mainloop()
```

Run the program and verify that a cyan rectangle is created and that you can move the rectangle using the left mouse button.



Canvas Class Modifications

The Canvas Class modifications are:

- Add a shape list
- Create an instance of the Mouse Class
- Add a redraw shapes method that iterates over the shape list and calls the shape update() method

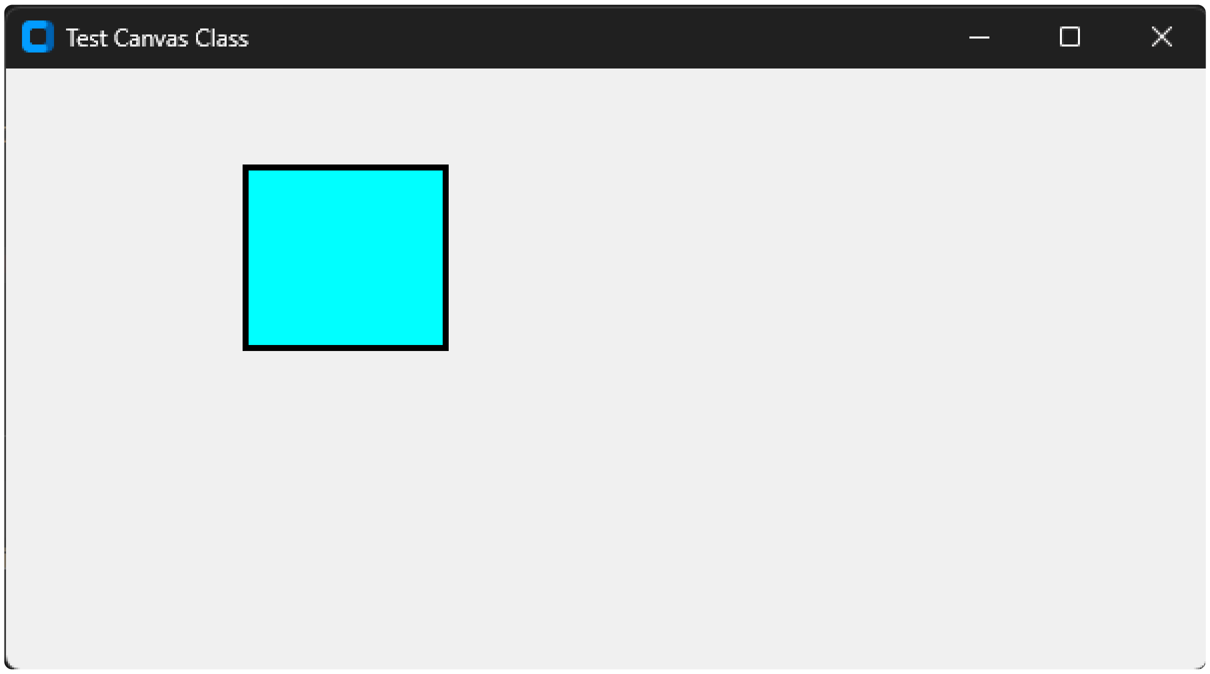
canvas.py

```
import customtkinter as ctk
from UI_LIb.mouse import Mouse


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

    def redraw_shapes(self):
        for s in self.shape_list:
            s.update()
```

We can add a main test section to test the canvas class before it is integrated into the main application.

```
. . .

if __name__ == "__main__":
    import customtkinter as ctk
    from Shape_Lib import Rectangle
    from UI_LIb import Mouse, Canvas

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Mouse Class")

    canvas = Canvas(app)
    mouse = Mouse(canvas)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    rect = Rectangle(canvas, 10, 10, 110, 100)
    rect.create()
    canvas.shape_list.append(rect)

    app.mainloop()
```

Run the program and confirm that a cyan rectangle is created and that you can move it with the left mouse button.



Shape Selection

We need a visual indicator that a shape is selected. We will use a red rectangle that is 5 px larger than the enclosed shape as a simple selector, for now.

Selection rules:

- Select a shape by left click on the shape
- Unselect all shapes by left click on the canvas, not on a shape
- Select more than one shapes by clicking on each shape

In the Rectangle Class, add a new attribute called `self.is_selected` which is a boolean variable and set it to False. Also, add a `self.sel_id` attribute and set it to None. The sel_id will store the index of the selection rectangle so it can be updated in the update() method. rectangle.py

```python
class Rectangle:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None
        self.sel_id = None  # Added
        self.is_selected = False  # Added

. . .
```

In the rectangle create() method create a red rectangle with coordinates 5 px larger than the original rectangle with transparent fill, outline color of red, and width of 2 px. Store the selector id in `self.sel_id`. Change the state of the selector rectangle to 'hidden' using `canvas.itemconfig()`.

```
. . .

   def create(self):
        """Create the shape once!"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)

        self.sel_id = self.canvas.create_rectangle(self.x1-5, self.y1-5,
self.x2+5, self.y2+5, fill=None,  # Added
                                                       outline="red", width=2)
        self.canvas.itemconfig(self.sel_id, state='hidden')  # Added

. . .
```

Modify the rectangle update() method to update the `coords` of the selection rectangle and change the state to 'normal' if selected or 'hidden' if not selected.

```
. . .

    def update(self):
        """Update the shape, don't recreate it!"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        # Update the selector position
        self.canvas.coords(self.sel_id, self.x1 - 5, self.y1 - 5, self.x2 + 5,
self.y2 + 5)
        if self.is_selected:
            self.canvas.itemconfig(self.sel_id, state='normal')
        else:
            self.canvas.itemconfig(self.sel_id, state='hidden')

. . .
```

In the Mouse Class, modify the select_hit_test() method to set the selected shape is_selected variable to True and redraw the shapes then return to the calling method. If the for loop does not return a shape, then no shape is selected (i.e. the user clicked on the empty portion of the canvas) and we set the selected shape to none and call the unselect all method. Add the unselect all method which iterates over the shape list and sets each shape is_selected variable to False.

```
. . .

def select_hit_test(self, x, y):
    for s in self.canvas.shape_list:
        if s.x1 <= x <= s.x2 and s.y1 <= y <= s.y2:
            self.selected_shape = s
            s.is_selected = True  # Added
            self.canvas.redraw_shapes()  # Added
            return  # Added

    # No shape hit - unselect all
    self.selected_shape = None  # Added
    self.unselect_all()  # Added

def unselect_all(self):  # Added new method
    # print("Unselect all")
    for s in self.canvas.shape_list:
        s.is_selected = False
        self.canvas.redraw_shapes()

. . .
```

Run the program and check that all selection rules defined above are satisfied.

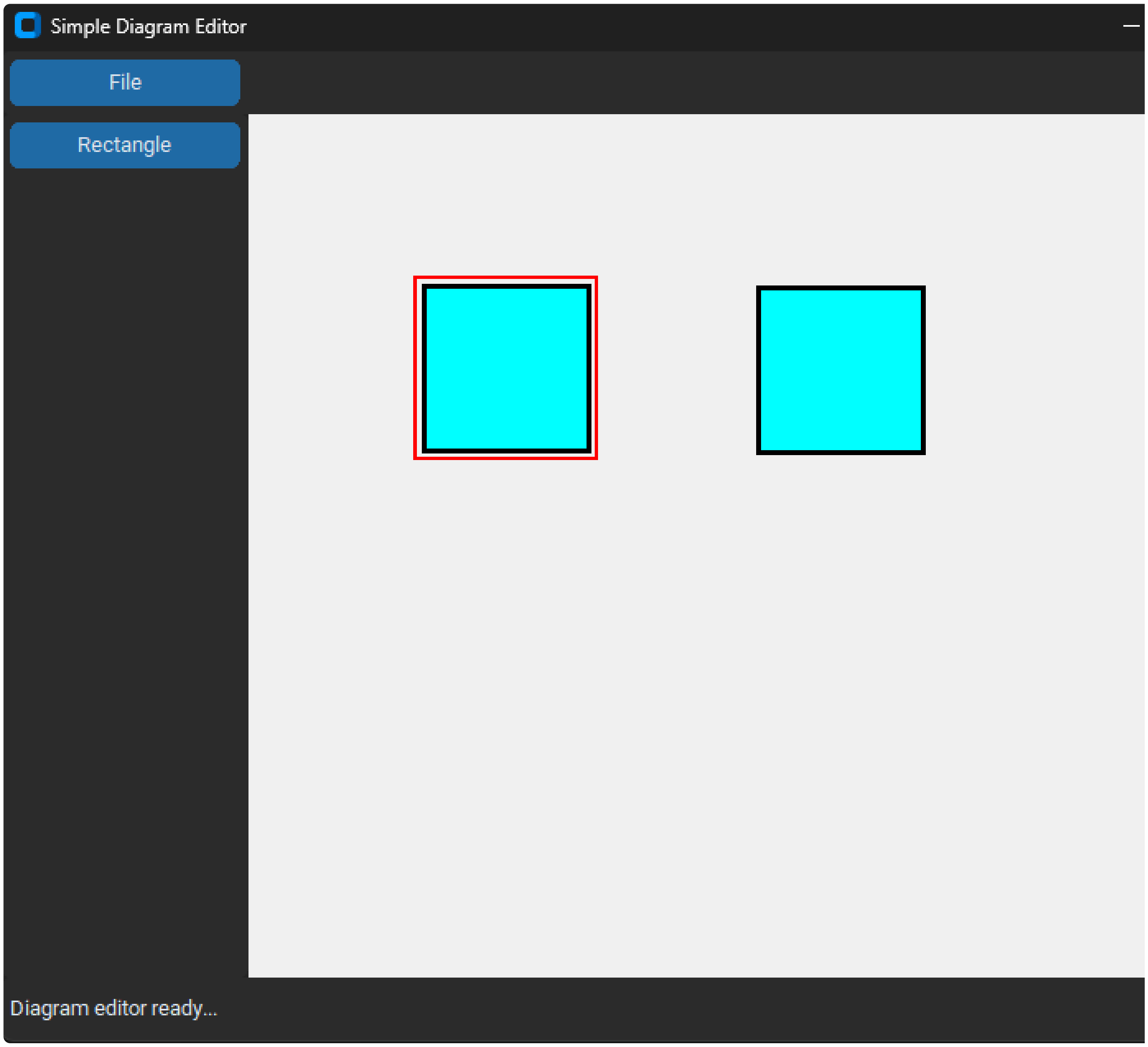


Diagram editor ready...

Shape Appearance Controls

Next we will add the shape appearance controls that control the selected shape's fill color, border color, and/or border width.

- Install `tktooltip` in a terminal type pip install tkinter-tooltip
- Create a custom subframe called Shape Appearance Frame in shape_appearance_frame.py
    - Create a custom frame for Fill Color containing Button and Color Label
    - Create a custom frame for Border Color containing Button and Color Label

    - Create a custom frame for Border Width containing Button with icon and width option menu
- Create tooltips for each of the controls
- Do not add it to UI_LIb/`__init__.py` to avoid circular import with Top Frame Class
- Add the subframe to the Top Frame

shape_appearance_frame.py

```python
import customtkinter as ctk
from CTkColorPicker import *
from tktooltip import ToolTip
from PIL import Image


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

        self.fill_label = None
        self.border_label = None

        self.init_fill_color_control(self)
        self.init_border_color_control(self)
        self.init_border_width_control(self)

    def init_fill_color_control(self, shape_appearance_frame):
        # Fill color frame
        fill_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
        fill_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add fill color picker
        def ask_fill_color():
            pick_color = AskColor()  # open the color picker
            color = pick_color.get()  # get the color string
            self.fill_label.configure(fg_color=color)
            for item in self.canvas.shape_list:
                if item.is_selected:
                    self.canvas.mouse.selected_shape.fill_color = color
                    self.canvas.redraw_shapes()

        fill_color_button = ctk.CTkButton(fill_frame, text="Fill",
text_color="black", width=50,
                                          command=ask_fill_color)
        fill_color_button.pack(side=ctk.LEFT, padx=5, pady=5)
        self.fill_label = ctk.CTkLabel(fill_frame, text="", width=30,
height=30, bg_color="white")
        self.fill_label.pack(side=ctk.LEFT, padx=5, pady=5)

        ToolTip(fill_color_button, msg="Fill color")

    def init_border_color_control(self, shape_appearance_frame):
        # Border color frame
        border_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
        border_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add border (outline) color picker
        def ask_border_color():
            pick_color = AskColor()  # open the color picker
            color = pick_color.get()  # get the color string
            self.border_label.configure(fg_color=color)
            for item in self.canvas.shape_list:
                if item.is_selected:
                    self.canvas.mouse.selected_shape.border_color = color
                    self.canvas.redraw_shapes()

        border_color_button = ctk.CTkButton(border_frame, text="Border",
text_color="black", width=50,
                                            command=ask_border_color)
        border_color_button.pack(side=ctk.LEFT, padx=10, pady=5)
        self.border_label = ctk.CTkLabel(border_frame, text="", width=30,
height=30, bg_color="white")
        self.border_label.pack(side=ctk.LEFT, padx=5, pady=5)

        ToolTip(border_color_button, msg="Border color")

    def init_border_width_control(self, shape_appearance_frame):
        # Border width frame
        width_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
        width_frame.configure(fg_color=("gray28", "gray28"))  # set frame color
        width_frame.pack(side=ctk.LEFT, padx=5, pady=5)
        my_image = ctk.CTkImage(light_image=Image.open
                                ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                                dark_image=Image.open
                                ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                                size=(24, 24))
        image_label = ctk.CTkLabel(width_frame, image=my_image, text="",
corner_radius=10)
        image_label.pack(side=ctk.LEFT)

        # Add OptionMenu to top frame
        def option_menu_callback(choice):
            for item in self.canvas.shape_list:
                if item.is_selected:
                    self.canvas.mouse.selected_shape.border_width = choice
                    self.canvas.redraw_shapes()

        option_menu = ctk.CTkOptionMenu(width_frame, values=["1", "2", "3",
"4", "5", "6", "7", "8", "9", "10"],
                                        width=32, command=option_menu_callback)
        option_menu.pack(side=ctk.LEFT)
        option_menu.set("3")

        ToolTip(option_menu, msg="Border width")
```

Top Frame Class

```
import customtkinter as ctk
from UI_LIb.shape_appearance_frame import ShapeAppearanceFrame  # Added


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

        # Add Top Frame widget here
        shape_frame = ShapeAppearanceFrame(self, self.canvas)  # Added
        shape_frame.pack(side=ctk.LEFT, padx=5, pady=5)  # Added
```

Notes:

- We need to use a more verbose import since both classes are in the same directory (package). We just need to specify the `directory.file_name` to import.

Main Simple Diagram Editor Application Class

```python
import customtkinter as ctk
from UI_LIb import Canvas, TopFrame, LeftFrame, BottomFrame


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self, self.canvas)  # Changed
        self.left_frame = LeftFrame(self, self.canvas)
        self.bottom_frame = BottomFrame(self)

        self.top_frame.pack(side=ctk.TOP, fill=ctk.BOTH)
        self.bottom_frame.pack(side=ctk.BOTTOM, fill=ctk.BOTH)
        self.left_frame.pack(side=ctk.LEFT, fill=ctk.BOTH)

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)


if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

Run the program, create a rectangle, select the rectangle, change the fill color, border color, and border width. Try creating two rectangles, selecting both rectangle, and change the appearance of both rectangles simultaneously! It did not work, why not?

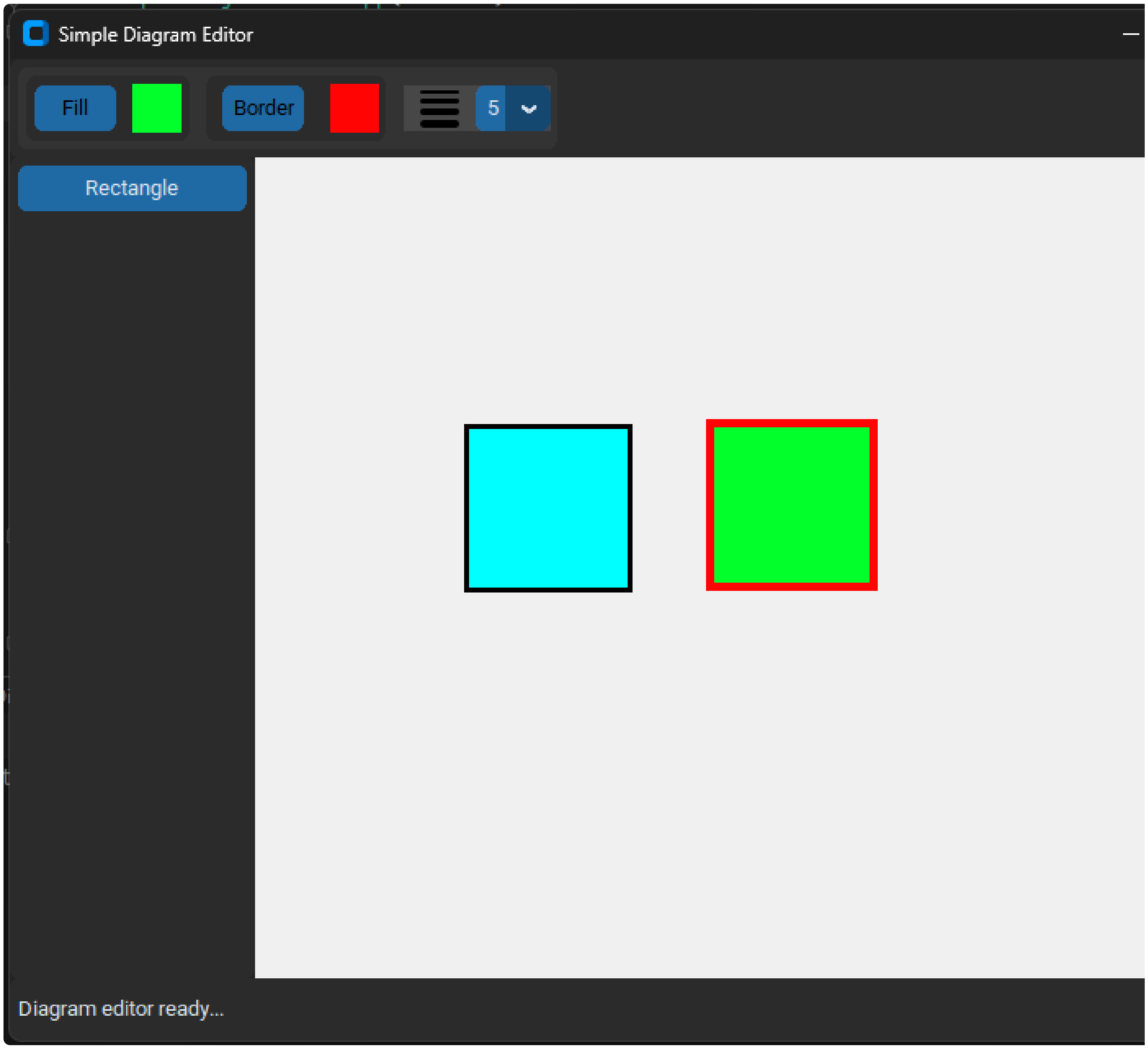
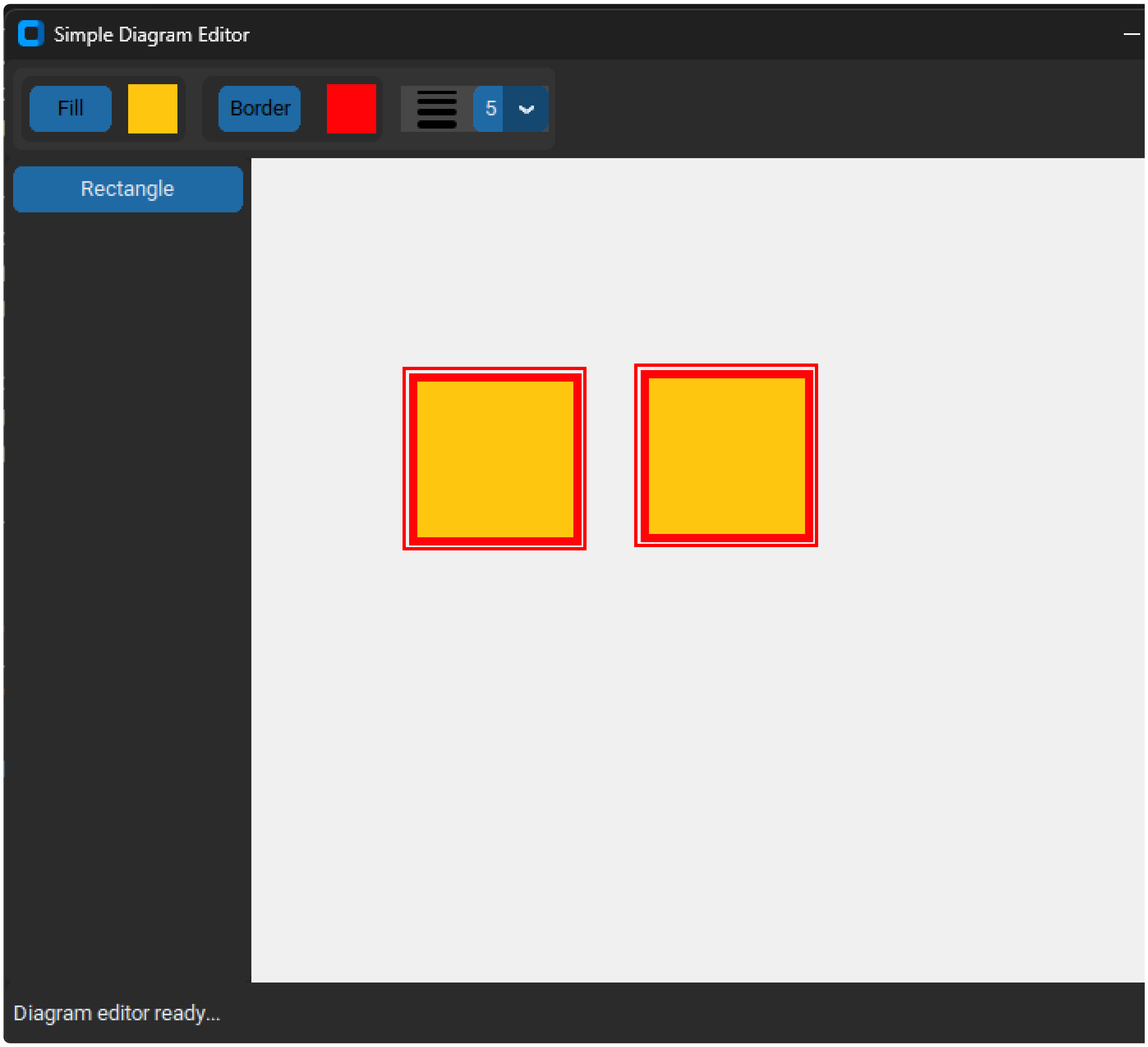


Diagram editor ready...

Instead of changing only the currently selected object, iterate over all shapes and change the appearance of all selected shapes using the `canvas.itemconfig()` method. This change can be made in the Shape Appearance Frame Class.

shape_appearance_frame.py

```
import customtkinter as ctk
from CTkColorPicker import *
from tktooltip import ToolTip
from PIL import Image


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

        self.fill_label = None
        self.border_label = None

        self.init_fill_color_control(self)
        self.init_border_color_control(self)
        self.init_border_width_control(self)

    def init_fill_color_control(self, shape_appearance_frame):
        # Fill color frame
        fill_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
        fill_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add fill color picker
        def ask_fill_color():
            pick_color = AskColor()  # open the color picker
            color = pick_color.get()  # get the color string
            self.fill_label.configure(fg_color=color)
            for item in self.canvas.shape_list:
                if item.is_selected:
                    item.fill_color = color
                    self.canvas.itemconfig(item.id, fill=color)  # Change

        fill_color_button = ctk.CTkButton(fill_frame, text="Fill",
text_color="black", width=50,
                                          command=ask_fill_color)
        fill_color_button.pack(side=ctk.LEFT, padx=5, pady=5)
        self.fill_label = ctk.CTkLabel(fill_frame, text="", width=30,
height=30, bg_color="white")
        self.fill_label.pack(side=ctk.LEFT, padx=5, pady=5)

        ToolTip(fill_color_button, msg="Fill color")

    def init_border_color_control(self, shape_appearance_frame):
        # Border color frame
        border_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
        border_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        # Add border (outline) color picker
        def ask_border_color():
            pick_color = AskColor()  # open the color picker
            color = pick_color.get()  # get the color string
            self.border_label.configure(fg_color=color)
            for item in self.canvas.shape_list:
                if item.is_selected:
                    item.border_color = color
                    self.canvas.itemconfig(item.id, outline=color)  # Change

        border_color_button = ctk.CTkButton(border_frame, text="Border",
text_color="black", width=50,
                                            command=ask_border_color)
        border_color_button.pack(side=ctk.LEFT, padx=10, pady=5)
        self.border_label = ctk.CTkLabel(border_frame, text="", width=30,
height=30, bg_color="white")
        self.border_label.pack(side=ctk.LEFT, padx=5, pady=5)

        ToolTip(border_color_button, msg="Border color")

    def init_border_width_control(self, shape_appearance_frame):
        # Border width frame
        width_frame = ctk.CTkFrame(shape_appearance_frame, width=150)
        width_frame.configure(fg_color=("gray28", "gray28"))  # set frame color
        width_frame.pack(side=ctk.LEFT, padx=5, pady=5)
        my_image = ctk.CTkImage(light_image=Image.open
                                ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                                dark_image=Image.open
                                ("D:/EETools/DiagramEditor/icons/line-
width.png"),
                                size=(24, 24))

        image_label = ctk.CTkLabel(width_frame, image=my_image, text="",
corner_radius=10)
        image_label.pack(side=ctk.LEFT)

        # Add OptionMenu to top frame
        def option_menu_callback(choice):
            for item in self.canvas.shape_list:
                if item.is_selected:
                    item.border_width = choice
                    self.canvas.itemconfig(item.id, width=choice)  # Change

        option_menu = ctk.CTkOptionMenu(width_frame, values=["1", "2", "3",
"4", "5", "6", "7", "8", "9", "10"],
                                        width=32, command=option_menu_callback)
        option_menu.pack(side=ctk.LEFT)
        option_menu.set("3")
        ToolTip(option_menu, msg="Border width")
```



That works for multiple selections.

Rotate Shape with Keyboard

Add a new Keyboard Class to the UI_Lib package. The selected shape can be rotated 90 degrees by pressing the 'r' key on the keyboard.

keyboard.py

```
class Keyboard:
    def __init__(self, parent, canvas):
        """Class to manage keyboard events"""
        self.parent = parent
        self.canvas = canvas
        self.selected_shape = None

        # Declare keyboard bindings
        self.parent.bind('<r>', self.rotate_shape)

    def rotate_shape(self, _event):
        for s in self.canvas.shape_list:
            if s.is_selected:
                s.rotate()
        self.canvas.redraw_shapes()
```

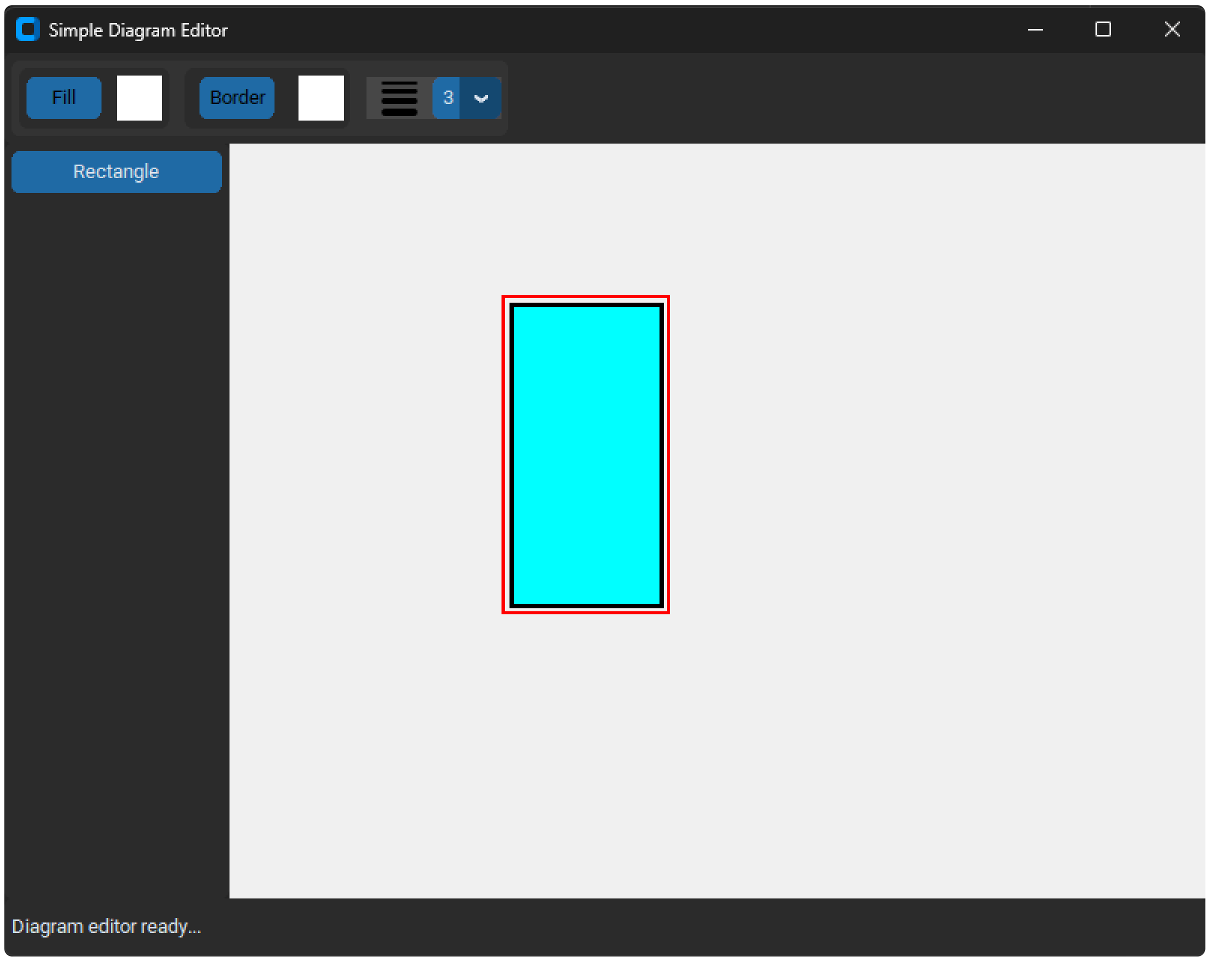
We need to add a rotate() method to the rectangle class.
rectangle.py modifications

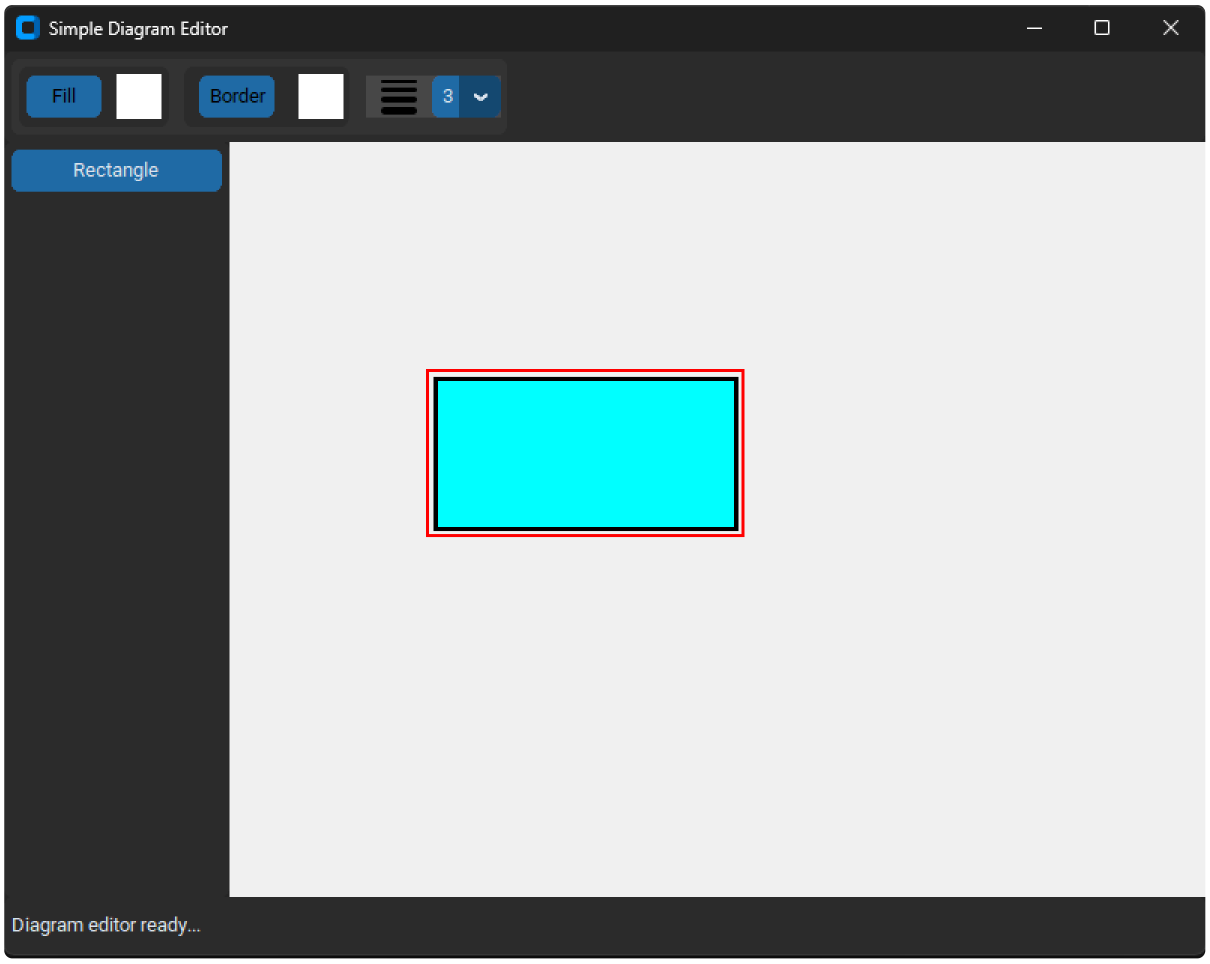
```
. . .

    def rotate(self):  # Added new method
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - h/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

    def __repr__(self):
        return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))

. . .
```

The keyboard needs to be instantiated in the Main Application File
simple_diagram_editor.py modifications

```
. . .

        self.canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

        self.keyboard = Keyboard(self, self.canvas)  # Added new


if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

In the rectangle class, I modified the size to 100x200 px. Run the program, create a rectangle, select the rectangle, press the 'r' key and verify that the rectangle rotates to 200x100.


Simple Diagram Editor
Fill
Border
3
Rectangle
Diagram editor ready...


Simple Diagram Editor
Fill
Border
3
Rectangle
Diagram editor ready...

This method works well for symmetric shapes. Non-symmetric shapes such as a triangle will need more work to define the shape at each rotation position. Text and images have a simple rotation feature. Lines will not be rotated because they can be resized to any rotation.

Draw Shape with Mouse

Currently, the program creates a rectangle with a fixed size and location on the canvas. In this section, we will modify the program to allow the user to select a shape from the Left Frame Menu and "draw" the shape with the mouse at any desired location and by dragging the left mouse button to any desired size. The Mouse Class has bindings for moving shapes with the mouse. We will add new bindings and event handlers to draw the shape and modify the button handler methods in the Left Frame Class to use the new draw bindings.

In the Mouse Class initializer, move the three mouse bindings to a method called `move_bind_mouse_events()`.

```python
from Helper_Lib.point import Point


class Mouse:
    def __init__(self, a_canvas):
        """Class to manage mouse events"""
        self.canvas = a_canvas
        self.selected_shape = None

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

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

. . .
```

Create a new method to unbind all mouse events and add a call to `unbind_mouse_events()` in the `move_bind_mouse_events()` method.

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

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

Next, add new mouse bindings for the draw event handlers.

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

Implement the three draw event handlers: `draw_left_down()`, `draw_left_drag()`, and `draw_left_up()` to draw a rectangle when called. In the draw left down event handler, we simply capture the start x, y coordinate which is the location of the mouse click on the canvas and create a rectangle at the start x, y coordinates. In draw left drag, we assume that the upper-left corner (x1, y1) of the rectangle is set to the start x, y coordinate then we set the lower-right corner (x2, y2) coordinates to the mouse drag position and update the shape. This gives the effect that we are 'drawing' the shape with the mouse. The draw left down method, initializes the current shape variables to None, unbinds the mouse events, and sets the mouse bindings to move mouse events.
mouse.py

```
. . .

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

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y
        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.canvas.shape_list.append(self.current_shape_obj)

    def draw_left_drag(self, event):
        if self.current_shape_obj:
            shape = self.current_shape_obj
            x, y = event.x, event.y
            shape.x1, shape.y1 = self.start.x, self.start.y
            shape.x2, shape.y2 = x, y
            self.canvas.redraw_shapes()

    def draw_left_up(self, _event):
        self.current_shape = None
        self.current_shape_obj = None
        self.unbind_mouse_events()
        self.move_bind_mouse_events()

. . .
```

In the Rectangle class, add a statement in the class initializer to create itself when the Rectangle object is created.

```
from Helper_Lib import Point


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

        self.id = None
        self.sel_id = None
        self.is_selected = False

        self.create()  # Added
    def create(self):
        """Create the shape once!"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)

        self.sel_id = self.canvas.create_rectangle(self.x1-5, self.y1-5,
self.x2+5, self.y2+5, fill=None,
                                                   outline="red", width=2)
        self.canvas.itemconfig(self.sel_id, state='hidden')
```

Run the program, select the Rectangle button from the Shape Menu in the Left Frame. Nothing happens. Click on the canvas and drag the mouse to create a rectangle. Draw a second rectangle and verify that you can move it, change the fill color, border color, and border width.

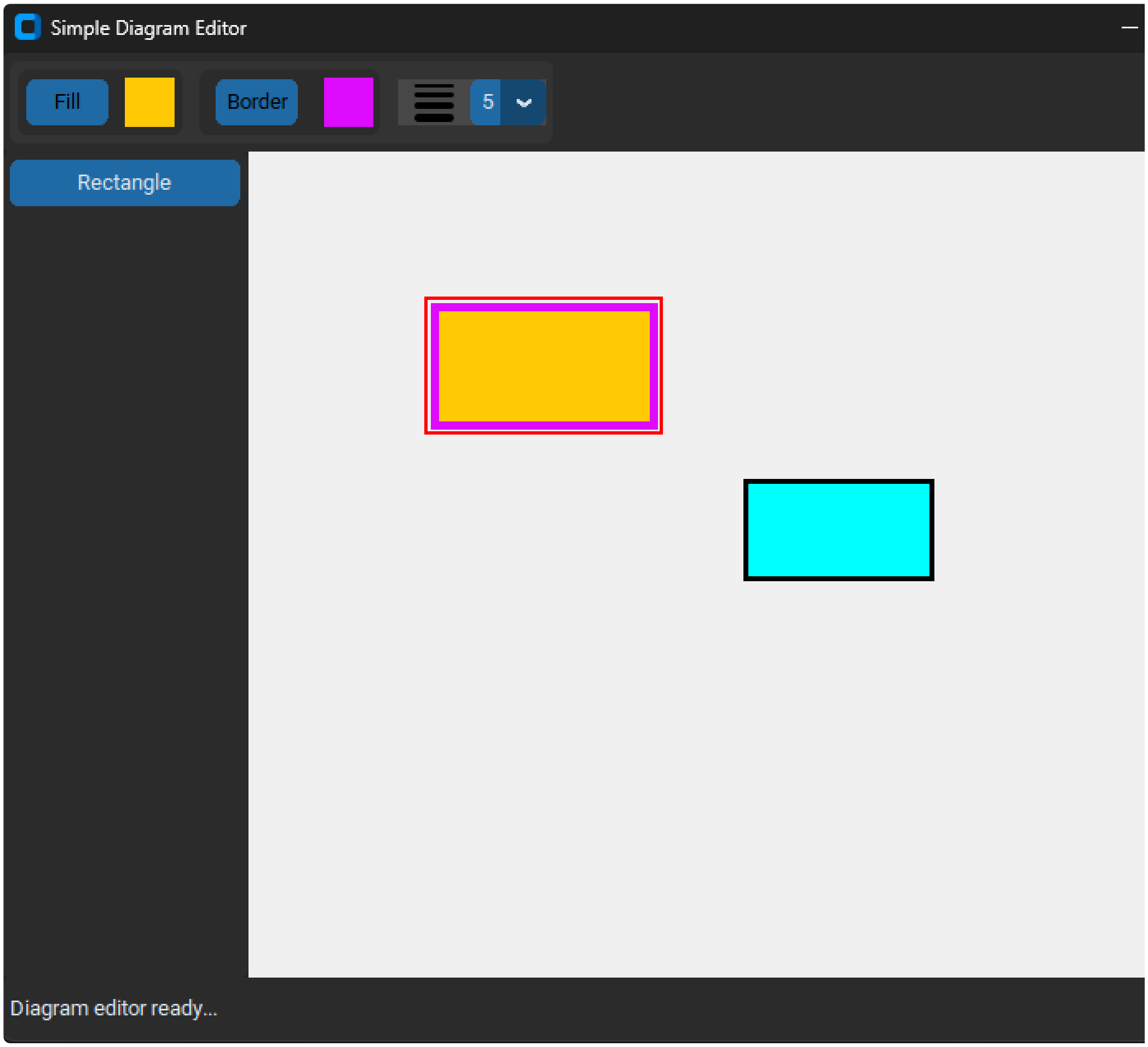


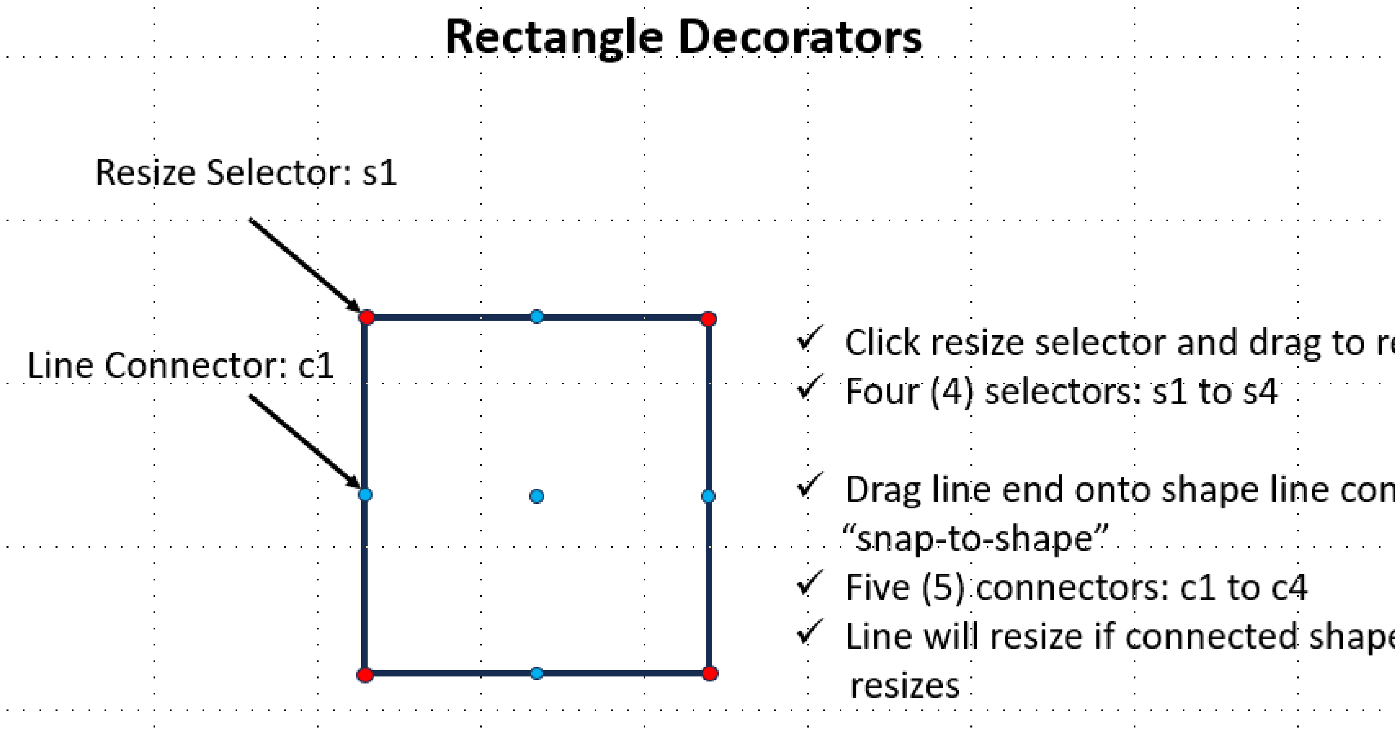
Diagram editor ready...

We can now draw shapes with the mouse.

Resize Shape with Mouse

We want to add a program "feature" that allows the user to resize an existing shape with the mouse. The user selects a corner of the shape and "drags" it with the left mouse button to resize the shape. We will add new mouse bindings and event handlers to resize the shape. How do we know that the user wants to resize the shape? We will modify the shape decorators to add selectors which are red ovals and if the user clicks on one of the selectors, we bind the mouse to the resize event handlers.

The following figure shows the two type of rectangle decorators we will need: "selectors" for resize mouse events and "connectors" for line connections to the shape (future capability).


Rectangle Decorators
Resize Selector: s1
Line Connector: c1
✓ Click resize selector and drag to r
✓ Four (4) selectors: s1 to s4
✓ Drag line end onto shape line con
"snap-to-shape"
✓ Five (5) connectors: c1 to c4
✓ Line will resize if connected shap
resizes

We will create classes for each of the decorator types.

Selector Class

Create a new file called selector.py in the Shape_Lib. Create a new class called Selector and initialize it with the canvas, a name, and x, y coordinates. Add an attributes called radius and set it to 5 px.

```
class Selector:
    def __init__(self, canvas, name, x, y):
        self.canvas = canvas
        self.name = name
        self.x = x
        self.y = y

        self.radius = 5
        self.id = None
```

Add a create() method that creates a red oval with outline color of black and width of 2 px. Call create() from the initializer.

```
class Selector:
    def __init__(self, a_canvas, name, x, y):
        self.canvas = a_canvas
        self.name = name
        self.x = x
        self.y = y

        self.radius = 5
        self.id = None
        self.create()

    def create(self):
        """Create the shape here"""
        sel_points = [self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius]
        self.canvas.create_oval(sel_points, fill="red", outline="black",
width=2)
```

Add an update() method to update the position of the selector.

```
   def update(self):
        """Update the shape here"""
        self.canvas.coords(self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius)
```

Add a selector hit test method that will be used to determine is the user clicks on it with the mouse.

```
    def selector_hit_test(self, event_x, event_y):
        x1, y1 = self.x - self.radius, self.y - self.radius
        x2, y2 = self.x + self.radius, self.y + self.radius
        if x1 <= event_x <= x2 and y1 <= event_y <= y2:
            return True
        else:
            return False
```

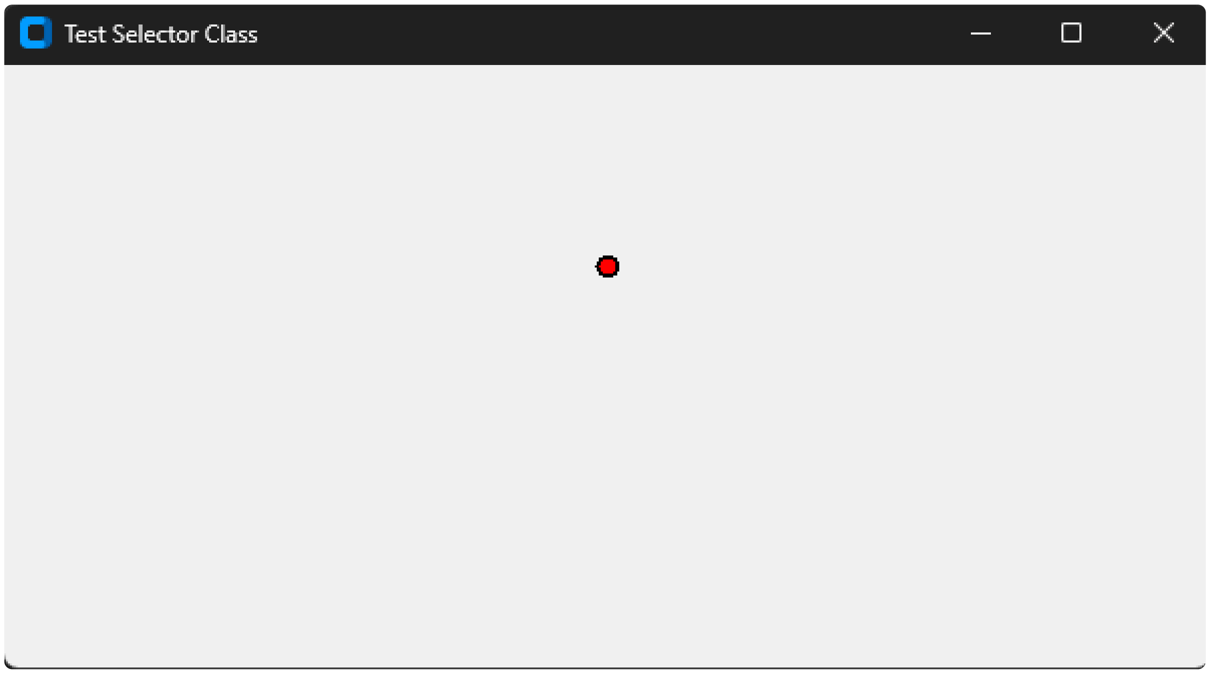
Add a `__repr__()` to override the print() method with a custom print string.

```
    def __repr__(self):
        return "Selector: " + self.name + " (" + str(self.x) + ", " +
str(self.y) + ")"
```

Add a main test section to test the selector locally before integrating it into the main program.

```
if __name__ == "__main__":
    import customtkinter as ctk

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Selector Class")

    canvas = ctk.CTkCanvas(app)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    sel = Selector(canvas, "test", 300, 100)
    print(sel)
    print("Hit test: ", sel.selector_hit_test(300, 100))

    app.mainloop()
```

Run the Selector Class program



Console Output

```
Selector: test (300, 100)
Hit test:  True
```

Here is the complete selector class without the test code.

```
class Selector:
    def __init__(self, a_canvas, name, x, y):
        self.canvas = a_canvas
        self.name = name
        self.x = x
        self.y = y

        self.radius = 5
        self.id = None
        self.create()

    def create(self):
        """Create the shape here"""
        sel_points = [self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius]
        self.canvas.create_oval(sel_points, fill="red", outline="black",
width=2)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.x - self.radius, self.y - self.radius, self.x +
self.radius, self.y + self.radius)

    def selector_hit_test(self, event_x, event_y):
        x1, y1 = self.x - self.radius, self.y - self.radius
        x2, y2 = self.x + self.radius, self.y + self.radius
        if x1 <= event_x <= x2 and y1 <= event_y <= y2:
            return True
        else:
            return False

    def __repr__(self):
        return "Selector: " + self.name + " (" + str(self.x) + ", " +
str(self.y) + ")"
```

Seems like a lot of work for one small red oval. We will see the power of this class as we add it to the Rectangle Class to define its selectors. We need to replace the simple red rectangle selector, with four selectors defined by our new Selector Class and position them at the four corners of the rectangle shape. First, we need to create the selectors, hide them until the rectangle is selected, and update their position if the rectangle is moved, rotated, or resized.

In the Rectangle Class, import the Selector Class
rectangle.py

```
from Helper_Lib import Point
from Shape_Lib.selector import Selector # Added
```

In the Rectangle Class initializer, create a selector list called `self.sel_list`. Also, create four select id variables and call a create selectors method that we will code soon.

```
def __init__(self, a_canvas, x1, y1, x2, y2):
    self.canvas = a_canvas
    self.x1 = x1
    self.y1 = y1
    self.x2 = x2
    self.y2 = y2
    self.fill_color = "cyan"
    self.border_color = "black"
    self.border_width = 3

    self.id = None
    self.is_selected = False
    self.sel_list = []  # Added

    self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None  #
Added

    self.create_shape()
    self.create_selectors() # Added
```

Modify the update() method to toggle the visibility of the selectors based on the is_selected variable.

```
    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')
```

Add the `create_selectors()` method to create four selectors, add them to the `sel_list`, and set visibility to off ('hidden').

```
    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')
```

Add a new method called `update_selectors()` which will update the position of all selectors based on the position of the parent rectangle.

```
    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x2, self.y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = self.x1, self.y2
        self.s4_id.update()
```

Add a new method called `resize()` that changes the rectangle coordinates based on the selector name and the mouse offset position.

```
    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            self.x1, self.y2 = x1, y2
```

Add a new method called `check_selector_hit()` that will check to see if a selector has been selected by the user.

```
    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None
```

No changes to the `rotate()` or `__repr__()` methods.

Here is the complete Rectangle Class.

```python
from Helper_Lib import Point
from Shape_Lib.selector import Selector


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

        self.id = None
        self.is_selected = False
        self.sel_list = []
        self.selector = None

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None

        self.create_shape()
        self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x2, self.y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = self.x1, self.y2
        self.s4_id.update()

    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - h/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            self.x1, self.y2 = x1, y2

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))


if __name__ == "__main__":
    import customtkinter as ctk
    from UI_LIb import Canvas

    app = ctk.CTk()
    app.geometry("600x300")
    app.title("Test Rectangle Class")

    canvas = Canvas(app)
    canvas.pack(side=ctk.RIGHT, fill=ctk.BOTH, expand=True)

    # Add widgets here
    canvas.mouse.current_shape = "rectangle"
    canvas.mouse.draw_bind_mouse_events()

    app.mainloop()
```

Run the main program, draw a rectangle, select the rectangle, and confirm that the selectors can be toggled on and off.

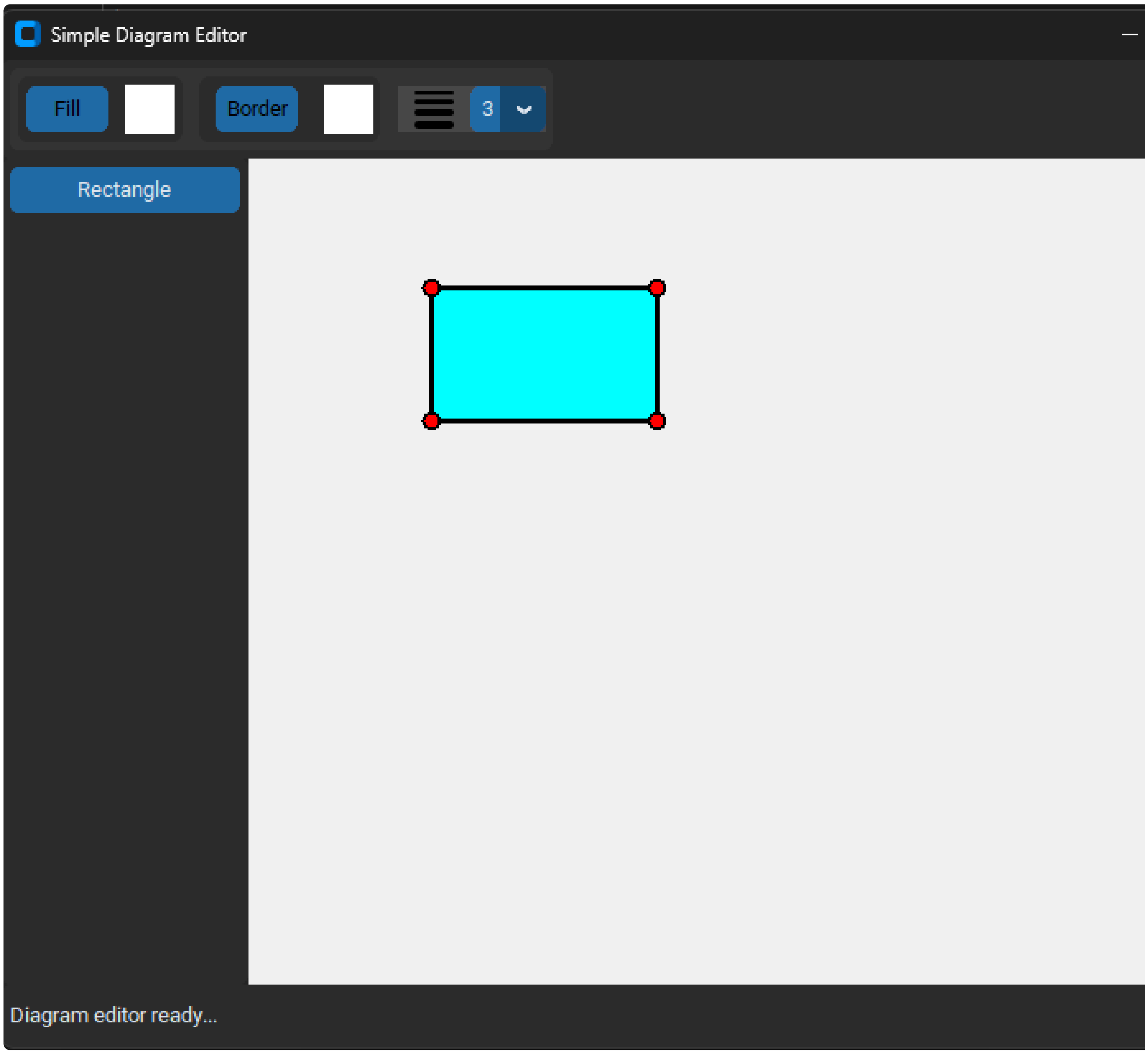


Diagram editor ready...

Next we need to modify the Mouse Class so that it can detect a "hit" on a selector and change to resize mouse bindings which allow the user to resize the rectangle by dragging the selector to a new position. This needs to work for any of the four selectors.

Modify the Mouse Class to add resize bindings and resize event handlers

```
. . .

    def draw_bind_mouse_events(self):
        self.canvas.bind("<Button-1>", self.draw_left_down)
        self.canvas.bind("<B1-Motion>", self.draw_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.draw_left_up)
    def bind_resize_mouse_events(self):  # Added new mouse bindings
        self.canvas.bind("<Button-1>", self.resize_left_down)
        self.canvas.bind("<B1-Motion>", self.resize_left_drag)
        self.canvas.bind("<ButtonRelease-1>", self.resize_left_up)

. . .
```

Add the resize event handlers to the Mouse Class. These handlers are very similar to the draw event handlers. The resize left down method captures the shape coordinates and calculates all x and y offsets. The resize left drag method creates an offsets list and calls the shape resize() method that we added to the Rectangle Class earlier and calls the canvas `redraw_shapes()` method. The resize left up method resets the offsets to 0 and changes the mouse bindings for move operations.

```
. . .

    def draw_left_up(self, _event):
        self.current_shape = None
        self.current_shape_obj = None
        self.unbind_mouse_events()
        self.move_bind_mouse_events()

    def resize_left_down(self, event):  # New resize event handler
        if self.selected_shape:
            shape = self.selected_shape
            x1, y1 = shape.x1, shape.y1
            x2, y2 = shape.x2, shape.y2
            self.offset1.x = event.x - x1
            self.offset1.y = event.y - y1
            self.offset2.x = event.x - x2
            self.offset2.y = event.y - y2

    def resize_left_drag(self, event):  # New resize event handler
        if self.selected_shape:
            offsets = [self.offset1.x, self.offset1.y, self.offset2.x,
self.offset2.y]
            self.selected_shape.resize(offsets, event)
            self.canvas.redraw_shapes()

    def resize_left_up(self, _event):  # New resize event handler
        self.offset1.x, self.offset1.y = 0, 0
        self.offset2.x, self.offset2.y = 0, 0
        self.canvas.mouse.unbind_mouse_events()
        self.canvas.mouse.move_bind_mouse_events()

. . .
```

Run the main program, draw a rectangle, select the rectangle to show the selectors, drag one or more of the selectors to confirm that you can resize the rectangle.

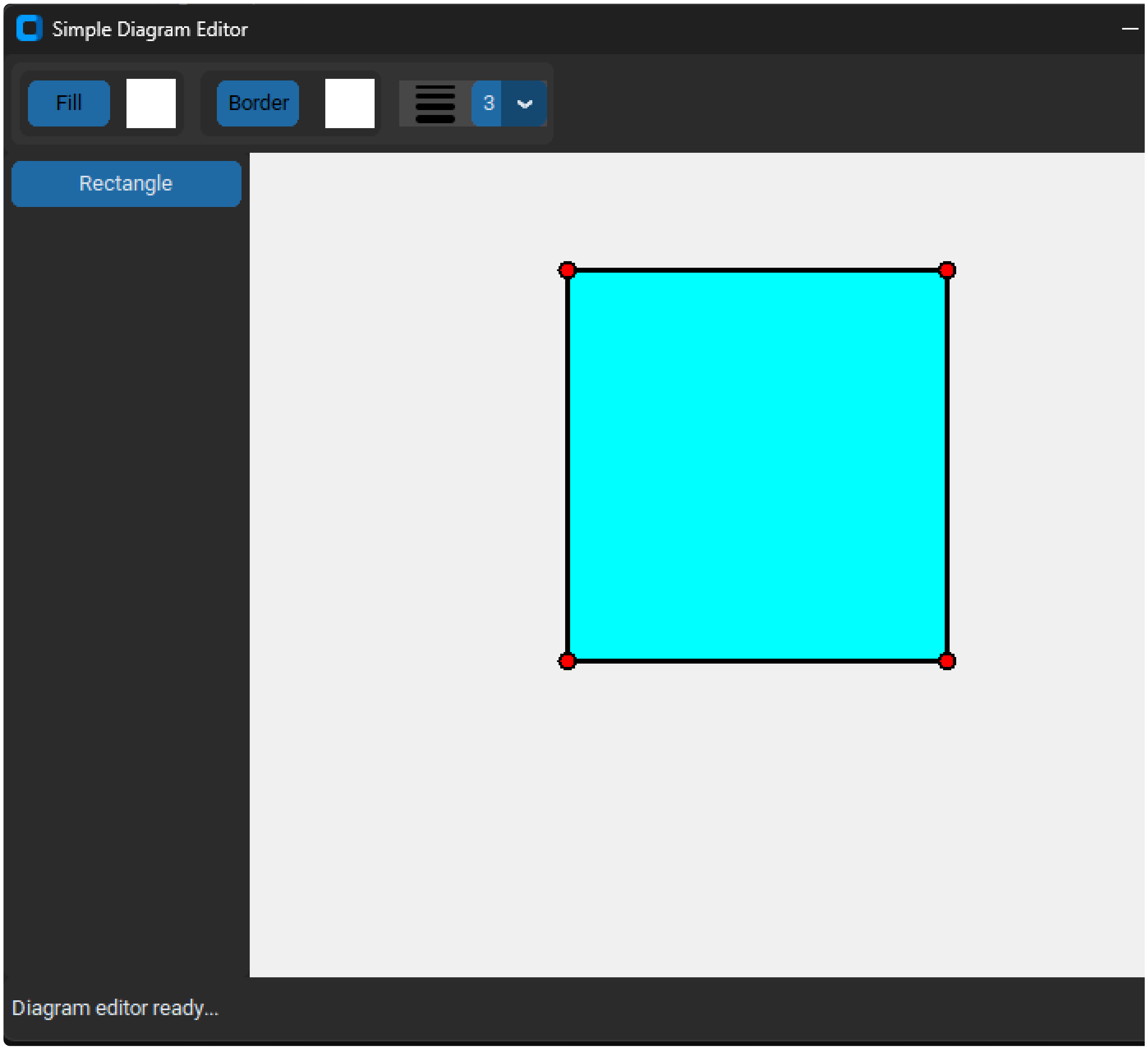


Diagram editor ready...

Refactor Test Code

Lets remove the test code section from:

- Rectangle Class
- Mouse Class
- Canvas Class
- Selector Class
- Point Class

Why remove the test code? The classes are fairly mature at this point and removing the test code means that we don't have to maintain it as the development progresses. We will add the test code section to any class that would benefit from local testing before integration into the main application.

After deleting the test code, run the program and confirm that it runs without errors.

Straight Line Class

We will create three line classes for the Diagram Editor App:

- Straight Line Class
- Segment Line Class - three segments
- Elbow Line Class - two segments

Lines are drawn between two endpoints: x1, y1 and x2, y2. Lines can be drawn on the canvas using `canvas.create_line()` method in a Line Class. In the Shape_Lib directory create a new file called straight_line.py.

In the new file, import the Selector Class

```
from Shape_Lib.selector import Selector
```

Name the class: `StraightLine` and initialize it with the canvas and the x1, y1, x2, y2 coordinates. Add the selector, id, is_selected and `sel_list` attributes. Set the fill color to black and width to 3. Recall that lines do not have an outline color. Create two selector id variables called `s1_id` and `s2_id`. Call the create shape and create selectors methods that we will add shortly.

```
class StraightLine:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.selector = None
        self.id = None
        self.is_selected = False
        self.sel_list = []

        self.fill_color = "black"
        self.border_width = 3

        self.s1_id, self.s2_id = None, None

        self.create_shape()
        self.create_selectors()
```

Add the `create_shape()` method that creates a Line Shape and stores a reference to the shape id.

```
    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
                                          fill=self.fill_color,
                                          width=self.border_width)
```

Add the update() method that updates the location, fill color, and border width of the line. It also call the `update_selectors()` method and sets the visibility of the selectors based on the is_selected variable.

```
def update(self):
    """Update the shape here"""
    self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
    self.canvas.itemconfig(self.id, fill=self.fill_color)
    self.canvas.itemconfig(self.id, width=self.border_width)

    self.update_selectors()
    if self.is_selected:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='normal')
    else:
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')
```

Add the `create_selectors()` method that creates two selectors:

- Name = 'begin' at the x1, y1 point
- Name = 'end' at the x2, y2 point
  It then adds the selector id's to the selector list and sets the visibility to hidden. Note that the line selectors will be shown when the line is selected.

```
    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')
```

Add the `update_selectors()` method that updates the position of the two selectors based on the line position coordinates.

```
    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()
```

Add a resize() method that move the end of the line associated with the selector name, either 'begin' or 'end'.

```
    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
```

Add a new method called `check_selector_hit()` that iterates over the selector list and calls the selector hit test. If hit, the method returns the selector otherwise it returns None.

```
def check_selector_hit(self, x, y):
    for sel in self.sel_list:
        if sel.selector_hit_test(x, y):
            return sel
    return None
```

Add a `__repr__()` method to provide a custom string when the line is printed.

```
    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)
```

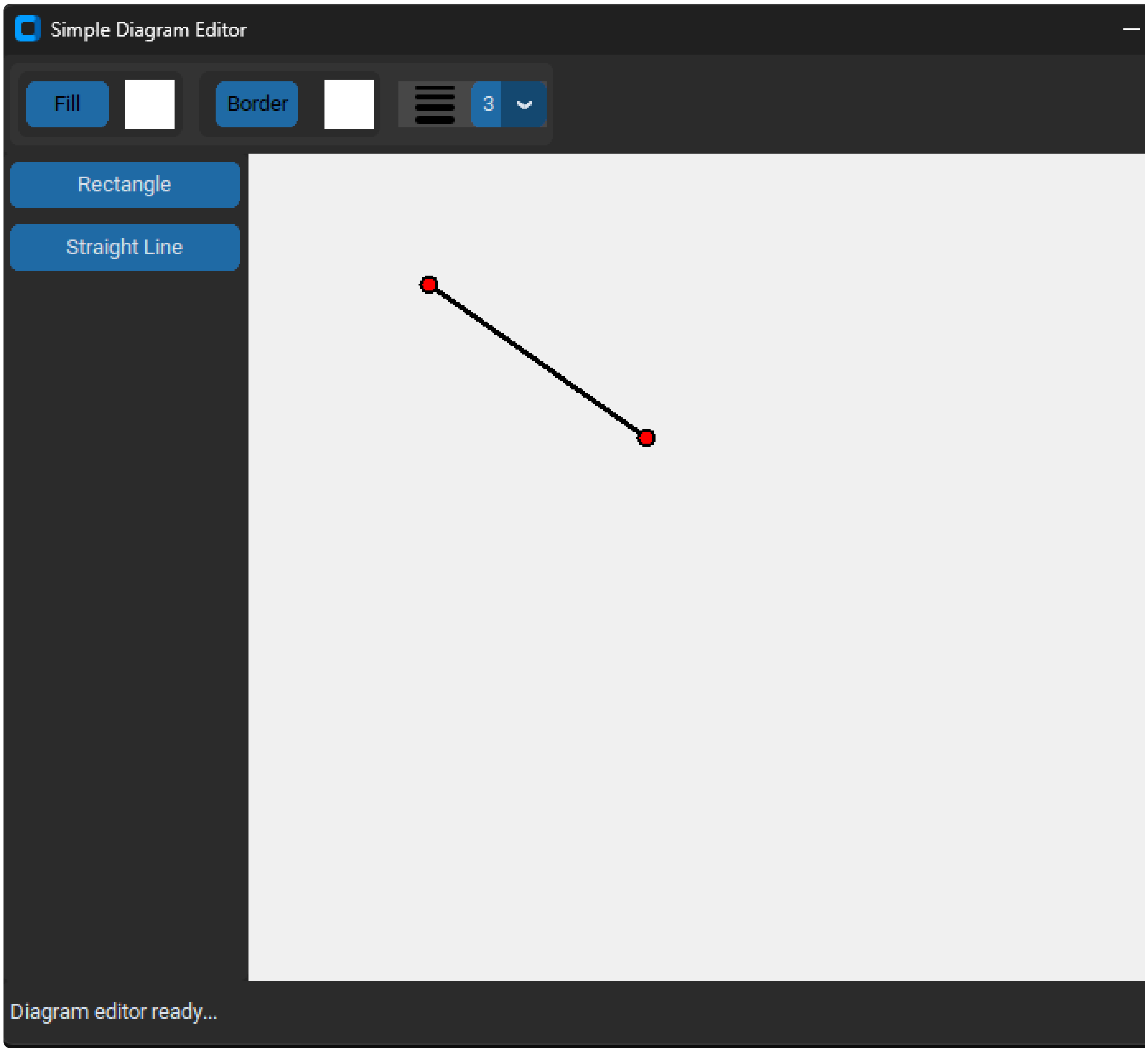
Here is the complete Straight Line Class.

```
from Shape_Lib.selector import Selector


class StraightLine:
    def __init__(self, a_canvas, x1, y1, x2, y2):
        self.canvas = a_canvas
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
        self.selector = None
        self.id = None
        self.is_selected = False
        self.sel_list = []
        self.fill_color = "black"
        self.border_width = 3

        self.s1_id, self.s2_id = None, None

        self.create_shape()
        self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_line(self.x1, self.y1, self.x2, self.y2,
                                          fill=self.fill_color,
                                          width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()
    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)
```

Modify the Left Frame Class to add a new button called Line and a button handler to set the current shape variable to "straight".
left_frame.py

```
import customtkinter as ctk


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

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

        straight_line_button = ctk.CTkButton(self, text="Straight Line",
command=self.create_straight_line)  # Added
        straight_line_button.pack(side=ctk.TOP, padx=5, pady=5)  # Added
    # Left frame button handlers
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.canvas.mouse.draw_bind_mouse_events()

    def create_straight_line(self):  # Added new method
        self.canvas.mouse.current_shape = "straight"
        self.canvas.mouse.draw_bind_mouse_events()
```

Run the main program, draw a line, and verify that you can move and resize the line with the mouse.



Connector Class

Shape Connectors

We will now implement the second shape decorator which is a connector represented graphically by a cyan oval. What is a connector used for? We want to add the program feature to draw lines between connectors on shapes such that if the shape is move, rotated, or resized the "connected" line will automatically resize.

Here is a summary of the changes needed to implement the connector feature:

- ✓ Create a new Connector Class
- ✓ Create a new Connection Class
- ✓ Modify the Rectangle Class to create and update Connectors
- ✓ Add a new `line_list` to the Rectangle Class to store "connections" to it
- ✓ Modify the mouse class for line drawing to check to see if a line is drawn from or to a connector on a shape. If there is a connector hit, add a "connection" to the parent shape's line list.
- ✓ Modify the Rectangle Class to move any connected lines if a shape is moved

Connector Class

Create a new file called connector.py in the Shape Library. The connector class is very similar to the selector class so I will show the code without explanation. If you need further explanation, look at the explanation in the Resize Shape with Mouse section.

```
class Connector:
    def __init__(self, a_canvas, name, _x, _y):
        self.canvas = a_canvas
        self.name = name
        self.x = _x
        self.y = _y

        self.radius = 5
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
self.radius,
                                              self.x + self.radius, self.y +
self.radius)

        self.id = None
        self.create()
    def create(self):
        """Create the shape here"""
        sel_points = [self.x1, self.y1, self.x2, self.y2]
        self.id = self.canvas.create_oval(sel_points, fill="cyan",
outline="black", width=2, tags='selector')

    def update(self):
        """Update the shape here"""
        self.x1, self.y1, self.x2, self.y2 = (self.x - self.radius, self.y -
self.radius,
                                              self.x + self.radius, self.y +
self.radius)
        sel_points = [self.x1, self.y1, self.x2, self.y2]
        self.canvas.coords(self.id, sel_points)

    def connector_hit_test(self, event_x, event_y):
        if self.x1 <= event_x <= self.x2 and self.y1 <= event_y <= self.y2:
            return True
        else:
            return False

    def __repr__(self):
        return ("Connector: " + self.name + " (" + str(self.x1) + ", " +
str(self.y1) + ")" +
                " (" + str(self.x2) + ", " + str(self.y2) + ")")
```

Add a new file called connection.py to the Shape Library. We will define a connection as the connector, line, and a line end.

```
class Connection:
    def __init__(self, conn_obj, line_obj, line_end):
        self.connector_obj = conn_obj
        self.line_obj = line_obj
        self.line_end = line_end      # "begin" or "end"

    def __repr__(self):
        return "Connection Object: " + self.connector_obj.name + \
               " Connection Object Location: " + str(self.connector_obj.x) + ",
" + str(self.connector_obj.y) + \
               " Line Object Points: " + str(self.line_obj.points) + \
               " Line End: " + self.line_end
```

Modify the Rectangle Class to add five connections to the center of the rectangle and the centers of each side.

```python
from Helper_Lib import Point
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector  # Added


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

        self.id = None
        self.is_selected = False
        self.is_drawing = False  # Added
        self.sel_list = []
        self.conn_list = []  # Added
        self.line_list = []  # Added
        self.selector = None

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
None, None, None, None  # Added

        self.create_shape()
        self.create_selectors()
        self.create_connectors()  # Added

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()  # Added
        if self.is_drawing:  # Added
            for c in self.conn_list:  # Added
                self.canvas.itemconfig(c.id, state='normal')  # Added
        else:  # Added
            for c in self.conn_list:  # Added
                self.canvas.itemconfig(c.id, state='hidden')  # Added

        self.move_connected_lines()  # Added

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x2, self.y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = self.x1, self.y2
        self.s4_id.update()

    def create_connectors(self):  # Added new method
        """Create connectors here"""
        # Calculate the shape geometry
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
        self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
        self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    def update_connectors(self):  # Added new method
        """Update the position of all connectors here"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id.x, self.c1_id.y = center.x, center.y
        self.c1_id.update()

        self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
        self.c2_id.update()

        self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
        self.c3_id.update()

        self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
        self.c4_id.update()

        self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
        self.c5_id.update()
    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - h/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            self.x1, self.y2 = x1, y2

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def check_connector_hit(self, x, y):  # Added new method
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None

    def move_connected_lines(self):  # Added new method
        for connection in self.line_list:
            for connector in self.conn_list:
                if connector == connection.connector_obj:
                    # print(connector, connection.line_obj, "Match")
                    if connection.line_end == "begin":
                        connection.line_obj.x1 = connector.x
                        connection.line_obj.y1 = connector.y
                    elif connection.line_end == "end":
                          connection.line_obj.x2 = connector.x
                          connection.line_obj.y2 = connector.y

    def __repr__(self):
        return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))
```

Modify the Left Frame Class to show connectors if the user draws a straight line.

```
import customtkinter as ctk


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

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

        straight_line_button = ctk.CTkButton(self, text="Straight Line",
command=self.create_straight_line)
        straight_line_button.pack(side=ctk.TOP, padx=5, pady=5)

    # Left frame button handlers
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_straight_line(self):
        self.canvas.mouse.current_shape = "straight"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def show_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = True
        self.canvas.redraw_shapes()

    def hide_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = False
        self.canvas.redraw_shapes()
```

Modify the Mouse Class to check for connector hits when drawing a line.

```
. . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y) # Added

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

. . .

    def draw_left_up(self, event):
        if isinstance(self.current_shape_obj, StraightLine):  # Added
            self.select_connector(self.current_shape_obj, "end", event.x,
event.y)  # Added
        self.current_shape = None
        self.current_shape_obj = None
        self.unbind_mouse_events()
        self.move_bind_mouse_events()

. . .

    def select_connector(self, line_obj, line_end, x, y):  # Added new method
```

```
        for shape in self.canvas.shape_list:
            conn = shape.check_connector_hit(x, y)
            if conn:
                if line_end == "begin":
                    line_obj.x1, line_obj.y1 = conn.x, conn.y
                elif line_end == "end":
                    line_obj.x2, line_obj.y2 = conn.x, conn.y
                a_conn = Connection(conn, line_obj, line_end)
                shape.line_list.append(a_conn)
                self.canvas.redraw_shapes()
                return
```

Run the program, draw two rectangles, draw one line between connectors on the two rectangle, select and move a rectangle, verify that the attached line automatically resizes to stay connected. Note that the connectors will be hidden when a new rectangle is drawn.

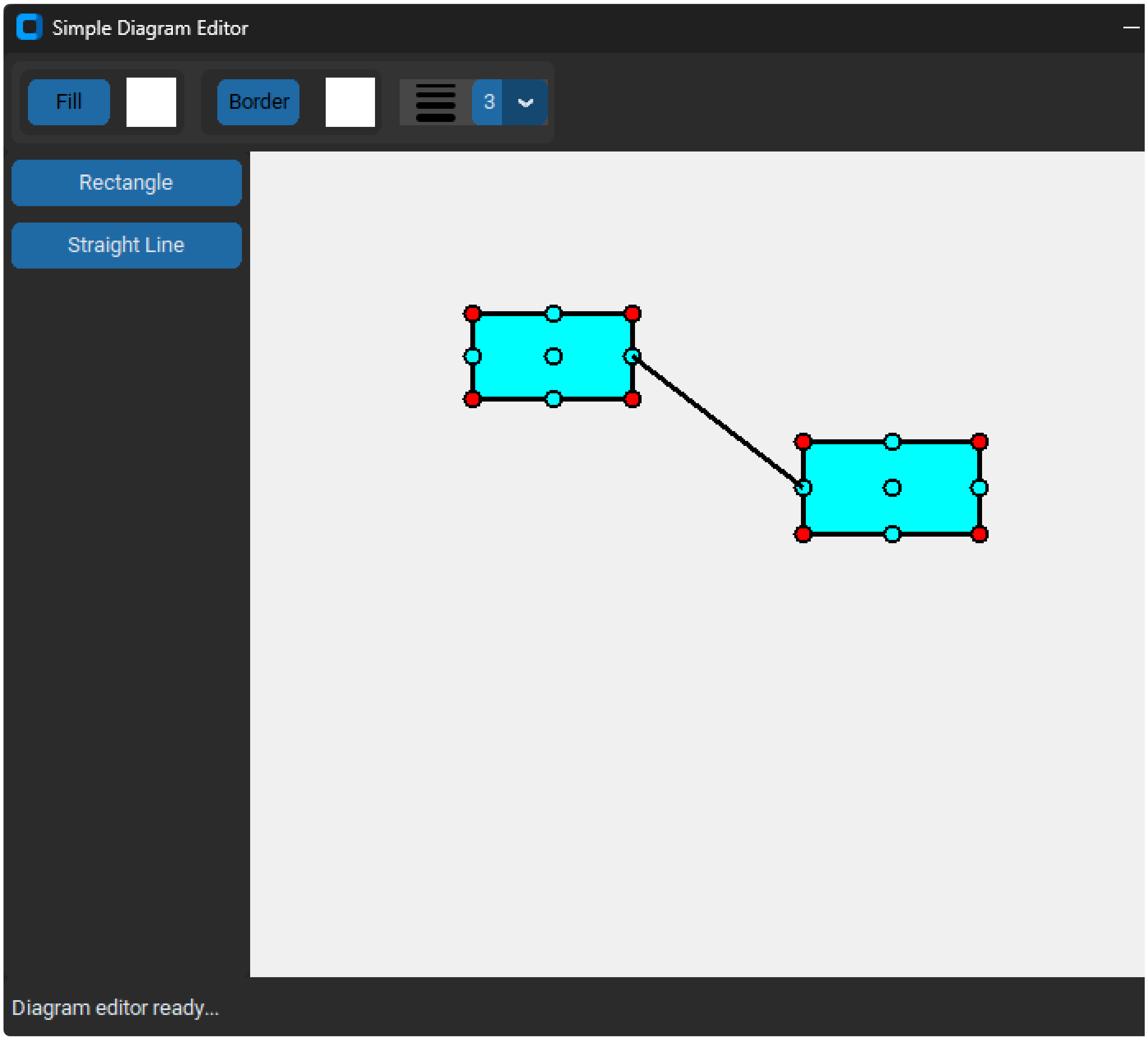


Diagram editor ready...

Shape Button Frames

In this section, we will create two custom frames for shape buttons and line button. The buttons will be images in a grid layout so it looks like a button array.

Shape Button Frame

The shape button frame will have five buttons for rectangle, oval, triangle, text, image. Create a new file called shape_button_frame.py in the UI_Lib.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image


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

        # Add left frame widgets here
        rect_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                size=(24, 24))
        rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,
command=self.create_rect)
        rect_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(rect_button, msg="Rectangle")

        oval_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                size=(24, 24))
        oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30)
        oval_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(oval_button, msg="Oval")

        tri_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                                size=(24, 24))
        tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30)
        tri_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
        ToolTip(tri_button, msg="Triangle")

        text_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                                size=(24, 24))
        text_button = ctk.CTkButton(self, text="", image=text_image, width=30)
        text_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(text_button, msg="Text")

        pic_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
                                size=(24, 24))
        pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30)
        pic_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(pic_button, msg="Image")

        # Shape button handlers
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def hide_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = False
        self.canvas.redraw_shapes()
```

Line Button Frame

The line button frame will have three buttons for straight line, segment line, and elbow line. Create a new file called line_button_frame.py in the UI_Lib.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image


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

        # Add left frame widgets here
        straight_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                        dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                        size=(24, 24))
        straight_button = ctk.CTkButton(self, text="", image=straight_image,
width=30,
                                        command=self.create_straight_line)
        straight_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(straight_button, msg="Straight Line")

        segment_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                        dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                        size=(24, 24))
        segment_button = ctk.CTkButton(self, text="", image=segment_image,
width=30)
        segment_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(segment_button, msg="Segment Line")

        elbow_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
                                size=(24, 24))
        elbow_button = ctk.CTkButton(self, text="", image=elbow_image,
width=30)
        elbow_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
        ToolTip(elbow_button, msg="Elbow Line")

    # Shape button handlers
    def create_straight_line(self):
        self.canvas.mouse.current_shape = "straight"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def show_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = True
        self.canvas.redraw_shapes()
```

Modify the Left Frame Class to use the new custom button frames.

```
import customtkinter as ctk
from UI_LIb.shape_button_frame import ShapeButtonFrame
from UI_LIb.line_button_frame import LineButtonFrame


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

        # Add left frame widgets here
        shape_button_frame = ShapeButtonFrame(self, self.canvas)
        shape_button_frame.pack(side=ctk.TOP, padx=5, pady=5)

        line_button_frame = LineButtonFrame(self, self.canvas)
        line_button_frame.pack(side=ctk.TOP, padx=5, pady=5)
```

Run the program, create two rectangles and a line and verify that there are no errors.

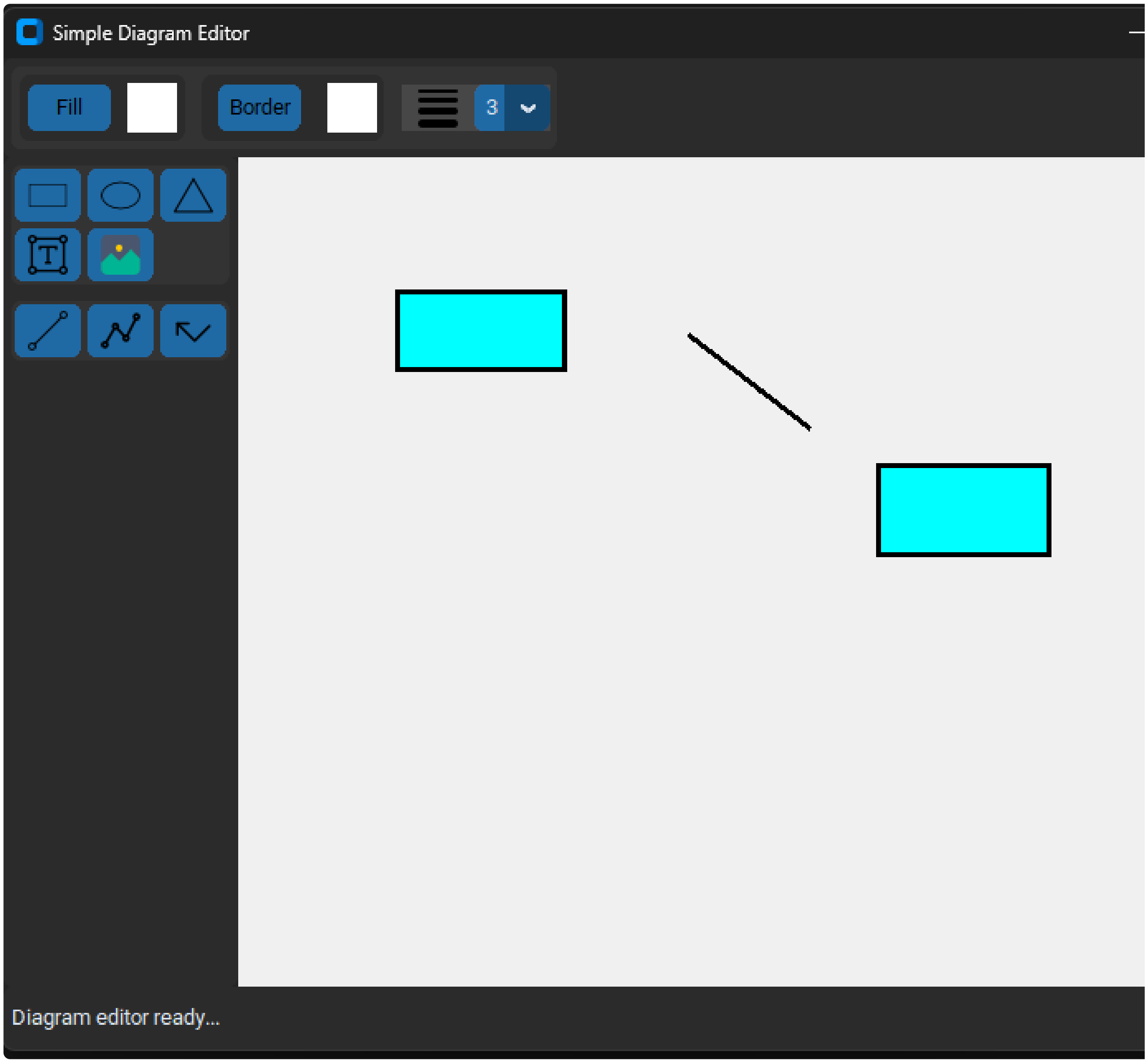


Diagram editor ready...

I like the custom button frames better that a vertical set of shape buttons. What do you think? Sometimes software programming is an art rather than a science.
Note that all the icons are available on GitHub.

Oval Class

We will model the Oval Class after the Rectangle Class.
oval.py

```
from Helper_Lib import Point
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector


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

        self.id = None
        self.is_selected = False
        self.is_drawing = False
        self.sel_list = []
        self.conn_list = []
        self.line_list = []
        self.selector = None

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_oval(self.x1, self.y1, self.x2, self.y2,
                                           fill=self.fill_color,
                                           outline=self.border_color,
                                           width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

        self.move_connected_lines()

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x2, self.y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = self.x1, self.y2
        self.s4_id.update()

    def create_connectors(self):
        """Create connectors here"""
        # Calculate the shape geometry
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
        self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
        self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    def update_connectors(self):
        """Update the position of all connectors here"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id.x, self.c1_id.y = center.x, center.y
        self.c1_id.update()

        self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
        self.c2_id.update()

        self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
        self.c3_id.update()

        self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
        self.c4_id.update()

        self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
        self.c5_id.update()

    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - h/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            self.x1, self.y2 = x1, y2

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def check_connector_hit(self, x, y):
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None

    def move_connected_lines(self):
        for connection in self.line_list:
            for connector in self.conn_list:
                if connector == connection.connector_obj:
                    # print(connector, connection.line_obj, "Match")
                    if connection.line_end == "begin":
                        connection.line_obj.x1 = connector.x
                        connection.line_obj.y1 = connector.y
                    elif connection.line_end == "end":
                        connection.line_obj.x2 = connector.x
                        connection.line_obj.y2 = connector.y

    def __repr__(self):
        return ("Oval: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))
```

Shape Base Class

Looking at the three shape classes: Rectangle, Oval, and `StraightLine` we can consider making a Shape Class as a base class with common attributes and methods. Create a new file in Shape_Lib called shape.py.

shape.py

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

        self.fill_color = "cyan"
        self.border_color = "black"
        self.border_width = 3

        self.id = None
        self.is_selected = False
        self.is_drawing = False
        self.sel_list = []
        self.conn_list = []
        self.line_list = []
        self.selector = None

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def check_connector_hit(self, x, y):
        for conn in self.conn_list:
            if conn.connector_hit_test(x, y):
                return conn
        return None
    def move_connected_lines(self):
        for connection in self.line_list:
            for connector in self.conn_list:
                if connector == connection.connector_obj:
                    # print(connector, connection.line_obj, "Match")
                    if connection.line_end == "begin":
                        connection.line_obj.x1 = connector.x
                        connection.line_obj.y1 = connector.y
                    elif connection.line_end == "end":
                        connection.line_obj.x2 = connector.x
                        connection.line_obj.y2 = connector.y
```

Rectangle Class derived from Shape Class

```
from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector


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

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_rectangle(self.x1, self.y1, self.x2,
self.y2,
                                               fill=self.fill_color,
                                               outline=self.border_color,
                                               width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

        self.move_connected_lines()

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x2, self.y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = self.x1, self.y2
        self.s4_id.update()

    def create_connectors(self):
        """Create connectors here"""
        # Calculate the shape geometry
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
        self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
        self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    def update_connectors(self):
        """Update the position of all connectors here"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id.x, self.c1_id.y = center.x, center.y
        self.c1_id.update()

        self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
        self.c2_id.update()

        self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
        self.c3_id.update()

        self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
        self.c4_id.update()

        self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
        self.c5_id.update()
    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - h/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            self.x1, self.y2 = x1, y2

    def __repr__(self):
        return ("Rectangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))
```

Run the program and verify that you can create a rectangle with no errors.

Refactored Oval Class

```
from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector


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

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id, self.c5_id = None,
None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape here"""
        self.id = self.canvas.create_oval(self.x1, self.y1, self.x2, self.y2,
                                           fill=self.fill_color,
                                           outline=self.border_color,
                                           width=self.border_width)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

        self.move_connected_lines()

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y1)
        self.s3_id = Selector(self.canvas, "s3", self.x2, self.y2)
        self.s4_id = Selector(self.canvas, "s4", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x2, self.y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = self.x1, self.y2
        self.s4_id.update()

    def create_connectors(self):
        """Create connectors here"""
        # Calculate the shape geometry
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
        self.c3_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
        self.c5_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id,
self.c5_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    def update_connectors(self):
        """Update the position of all connectors here"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id.x, self.c1_id.y = center.x, center.y
        self.c1_id.update()

        self.c2_id.x, self.c2_id.y = center.x, center.y - h/2
        self.c2_id.update()

        self.c3_id.x, self.c3_id.y = center.x + w/2, center.y
        self.c3_id.update()

        self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
        self.c4_id.update()

        self.c5_id.x, self.c5_id.y = center.x - w/2, center.y
        self.c5_id.update()

    def rotate(self):
        w, h = self.x2 - self.x1, self.y2 - self.y1
        center = Point(self.x1 + w / 2, self.y1 + h / 2)
        self.x1, self.y1 = center.x - h/2, center.y - w/2
        self.x2, self.y2 = center.x + h/2, center.y + w/2

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            self.x1, self.y2 = x1, y2

    def __repr__(self):
        return ("Oval: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))
```

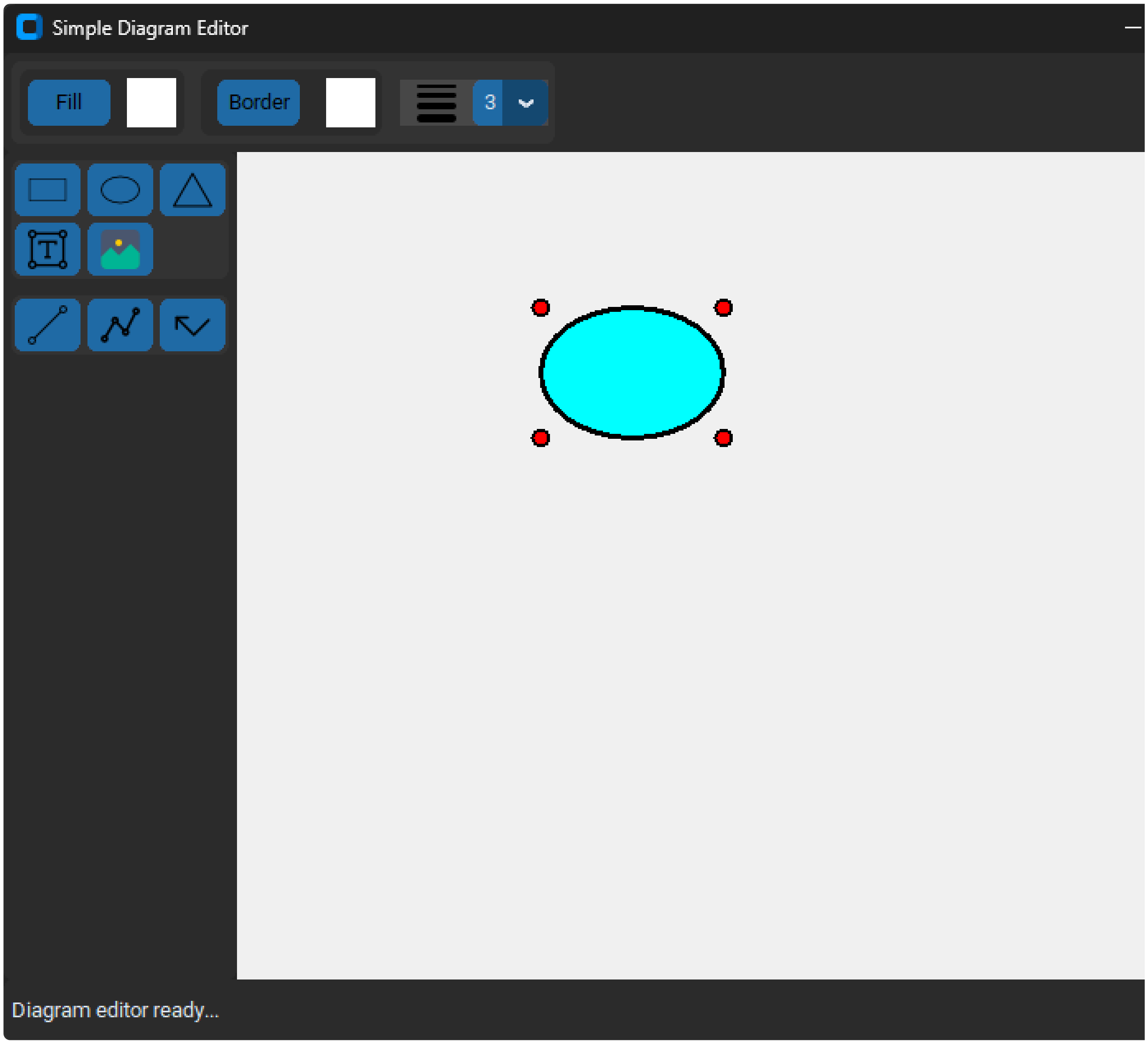
Update the Shape Button Frame Class to create an oval.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image


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

        # Add left frame widgets here
        rect_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                size=(24, 24))
        rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,
command=self.create_rect)
        rect_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(rect_button, msg="Rectangle")

        oval_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                size=(24, 24))
        oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30,
command=self.create_oval)  # Changed
        oval_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(oval_button, msg="Oval")
        tri_image = ctk.CTkImage(light_image=Image.open

("D:/EEToools/SimpleDiagramEditor/icons/triangle.png"),
                                dark_image=Image.open

("D:/EEToools/SimpleDiagramEditor/icons/triangle.png"),
                                size=(24, 24))
        tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30)
        tri_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
        ToolTip(tri_button, msg="Triangle")

        text_image = ctk.CTkImage(light_image=Image.open

("D:/EEToools/SimpleDiagramEditor/icons/text.png"),
                                dark_image=Image.open

("D:/EEToools/SimpleDiagramEditor/icons/text.png"),
                                size=(24, 24))
        text_button = ctk.CTkButton(self, text="", image=text_image, width=30)
        text_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(text_button, msg="Text")

        pic_image = ctk.CTkImage(light_image=Image.open

("D:/EEToools/SimpleDiagramEditor/icons/picture.png"),
                                dark_image=Image.open

("D:/EEToools/SimpleDiagramEditor/icons/picture.png"),
                                size=(24, 24))
        pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30)
        pic_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(pic_button, msg="Image")

        # Shape button handlers
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_oval(self):  # Added new method
        self.canvas.mouse.current_shape = "oval"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def hide_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = False
        self.canvas.redraw_shapes()
```

Update the mouse class to draw an oval.
mouse.py modifications

```
from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, StraightLine, Connection  # Changed

. . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "oval":  # Added
            self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)  # Added
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

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

. . .
```

Run the program and verify that you can create an oval with all the same features as the rectangle.



Triangle Class

The Triangle Class shape is created with a polygon with three points defining an isosceles triangle.
triangle.py

```
from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector
```

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

        self.s1_id, self.s2_id, self.s3_id = None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id = None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape once!"""
        w = self.x2 - self.x1
        self.points = [self.x1 + w / 2, self.y1, self.x2, self.y2, self.x1,
self.y2]
        self.id = self.canvas.create_polygon(self.points, fill=self.fill_color,
outline=self.border_color,
                                   width=self.border_width)

    def update(self):
        w = self.x2 - self.x1
        self.points = [self.x1 + w / 2, self.y1, self.x2, self.y2, self.x1,
self.y2]
        self.canvas.coords(self.id, self.points[0], self.points[1],
self.points[2], self.points[3],
                           self.points[4], self.points[5])
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

        self.move_connected_lines()

    def create_selectors(self):
        """Create four selectors at the corners here"""
        w = self.x2 - self.x1
        self.s1_id = Selector(self.canvas, "s1", self.x1 + w / 2, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y2)
        self.s3_id = Selector(self.canvas, "s3", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        w = self.x2 - self.x1
        self.s1_id.x, self.s1_id.y = self.x1 + w / 2, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = self.x1, self.y2
        self.s3_id.update()

    def create_connectors(self):
        """Create connectors here"""
        # Calculate the shape geometry
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x + w/4, center.y)
# Right Center
        self.c3_id = Connector(self.canvas, "c3", center.x - w/4, center.y)
# Left Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')
    def update_connectors(self):
        """Update the position of all connectors here"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id.x, self.c1_id.y = center.x, center.y
        self.c1_id.update()

        self.c2_id.x, self.c2_id.y = center.x + w/4, center.y
        self.c2_id.update()

        self.c3_id.x, self.c3_id.y = center.x - w/4, center.y
        self.c3_id.update()

        self.c4_id.x, self.c4_id.y = center.x, center.y + h/2
        self.c4_id.update()

    def __repr__(self):
        return ("Triangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))
```

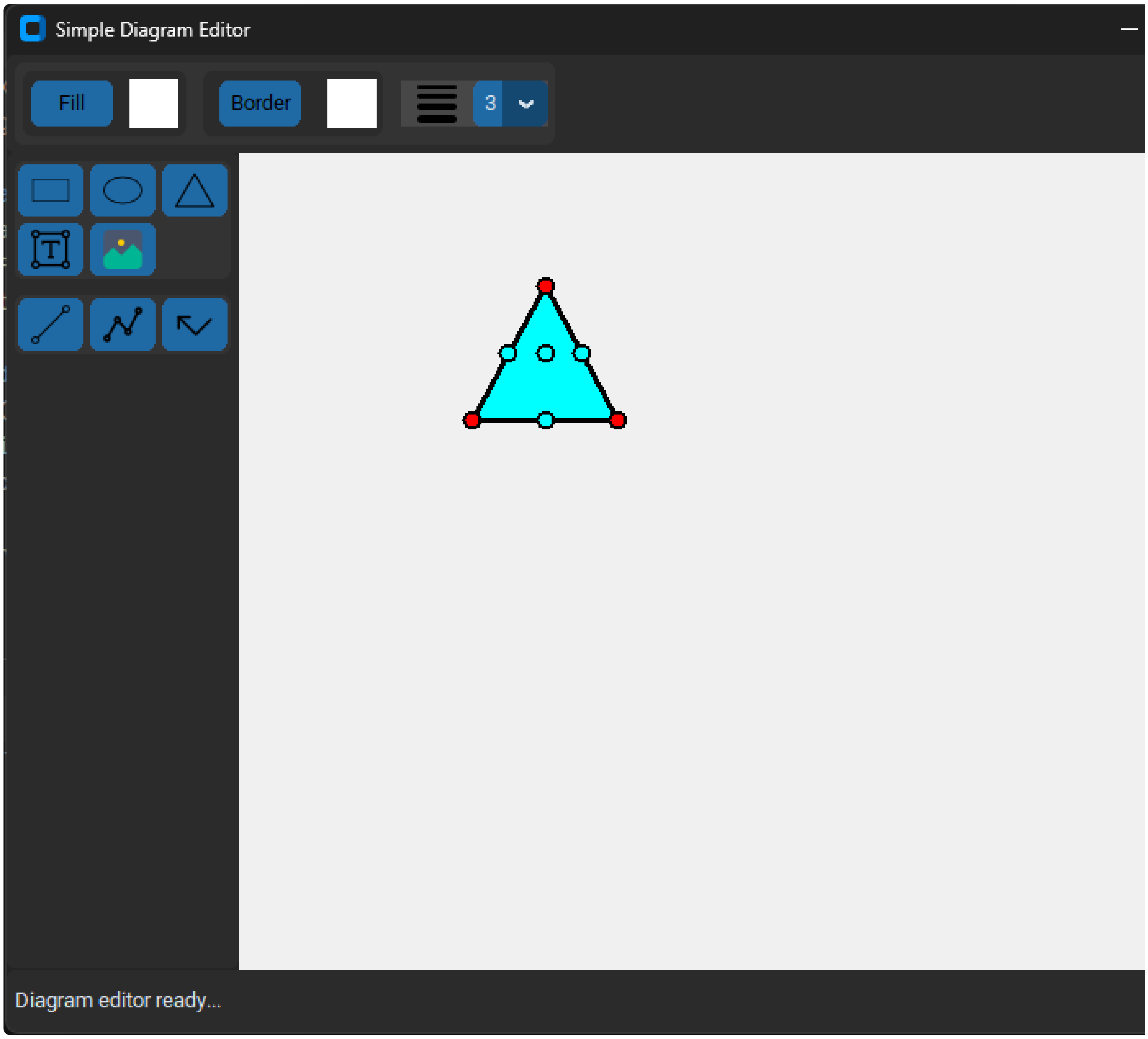
Update the Shape Button Frame Class to create an oval.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image


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

        # Add left frame widgets here
        rect_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                size=(24, 24))
        rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,
command=self.create_rect)
        rect_button.grid(row=0, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(rect_button, msg="Rectangle")

        oval_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                size=(24, 24))
        oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30,
command=self.create_oval)
        oval_button.grid(row=0, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(oval_button, msg="Oval")

        tri_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                                size=(24, 24))
        tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30,
command=self.create_tri) # Changed
        tri_button.grid(row=0, column=2, sticky=ctk.W, padx=2, pady=2)
        ToolTip(tri_button, msg="Triangle")

        text_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                                size=(24, 24))
        text_button = ctk.CTkButton(self, text="", image=text_image, width=30)
        text_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(text_button, msg="Text")

        pic_image = ctk.CTkImage(light_image=Image.open
("D:/EEToolS/SimpleDiagramEditor/icons/picture.png"),
                                dark_image=Image.open

("D:/EEToolS/SimpleDiagramEditor/icons/picture.png"),
                                size=(24, 24))
        pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30)
        pic_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(pic_button, msg="Image")

        # Shape button handlers
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

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

    def create_tri(self):  # Added new method
        self.canvas.mouse.current_shape = "tri"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def hide_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = False
        self.canvas.redraw_shapes()
```

Modify the Mouse Class to draw a triangle.

```
from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, StraightLine, Connection  #
Changed

. . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y
        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "oval":
            self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "tri":
            self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

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

. . .
```

Run the program and verify that you can draw a triangle with the same features as a rectangle.



Picture Class

The Picture Class will allow the user to add images from the file system to the diagram.
picture.py

```
from PIL import Image, ImageTk
import tkinter as tk

from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector
```

```python
class Picture(Shape):
    def __init__(self, canvas, x1, y1, x2=0, y2=0):
        super().__init__(canvas, x1, y1, x2, y2)
        self.a_image = None
        self.ph_image = None
        self.filename = "D:/EETools/DiagramEditor/images/hamburger.png"
        self.angle = 0
        self.bbox = None
        self.type = "picture"

        self.s1_id, self.s2_id, self.s3_id, self.s4_id = None, None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id = None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape here"""
        self.a_image = Image.open(self.filename)
        self.a_image = self.a_image.rotate(self.angle)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.id = self.canvas.create_image(self.x1, self.y1, anchor=tk.CENTER,
image=self.ph_image, tags="pics")
        self.bbox = self.canvas.bbox(self.id)

    def update(self):
        """Update the shape here"""
        self.canvas.coords(self.id, self.x1, self.y1)

        self.a_image = Image.open(self.filename)
        self.a_image = self.a_image.rotate(self.angle)
        self.ph_image = ImageTk.PhotoImage(self.a_image)
        self.canvas.itemconfig(self.id, image=self.ph_image)

        self.bbox = self.canvas.bbox(self.id)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')
        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

        self.move_connected_lines()

    def create_selectors(self):
        # Calculate position of selectors from current shape position
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]

        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "s1", x1, y1)
        self.s2_id = Selector(self.canvas, "s2", x2, y1)
        self.s3_id = Selector(self.canvas, "s3", x2, y2)
        self.s4_id = Selector(self.canvas, "s4", x1, y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id, self.s4_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        # Recalculate position of selectors from current shape position
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]

        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = x1, y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = x2, y1
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = x2, y2
        self.s3_id.update()

        self.s4_id.x, self.s4_id.y = x1, y2
        self.s4_id.update()

    def create_connectors(self):
        """Create connectors here"""
        # Calculate the shape geometry
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1+w/2, y1+h/2)

        self.c1_id = Connector(self.canvas, "c2", center.x, center.y - h/2)
# Top Center
        self.c2_id = Connector(self.canvas, "c3", center.x + w/2, center.y)
# Right Center
        self.c3_id = Connector(self.canvas, "c4", center.x, center.y + h/2)
# Bottom Center
        self.c4_id = Connector(self.canvas, "c5", center.x - w/2, center.y)
# Left Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    def update_connectors(self):
        """Update the position of all connectors here"""
        x1, y1, x2, y2 = self.bbox[0], self.bbox[1], self.bbox[2], self.bbox[3]
        w = x2 - x1
        h = y2 - y1
        center = Point(x1+w/2, y1+h/2)

        self.c1_id.x, self.c1_id.y = center.x, center.y - h/2
        self.c1_id.update()

        self.c2_id.x, self.c2_id.y = center.x + w/2, center.y
        self.c2_id.update()

        self.c3_id.x, self.c3_id.y = center.x, center.y + h/2
        self.c3_id.update()

        self.c4_id.x, self.c4_id.y = center.x - w/2, center.y
        self.c4_id.update()

    def rotate(self):
        """Calculate rotation angle"""
        self.angle += 90
        if self.angle > 270:
            self.angle = 0
        self.update()

    def resize(self, offsets, event):
        pass

    def __repr__(self):
        return ("Picture: x, y = " + "(" + str(self.x1) + ", " + str(self.y1) +
                "\nimage filename = " + self.filename)
```

shape_button_frame.py modifications.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image  # Added


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

        # Add left frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Shapes", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        rect_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/rectangle.png"),
                                size=(24, 24))
        rect_button = ctk.CTkButton(self, text="", image=rect_image, width=30,
command=self.create_rect)
        rect_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(rect_button, msg="Rectangle")

        oval_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/oval.png"),
                                size=(24, 24))
        oval_button = ctk.CTkButton(self, text="", image=oval_image, width=30,
command=self.create_oval)
```

```
        oval_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(oval_button, msg="Oval")

        tri_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/triangle.png"),
                                size=(24, 24))
        tri_button = ctk.CTkButton(self, text="", image=tri_image, width=30,
command=self.create_tri)
        tri_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)
        ToolTip(tri_button, msg="Triangle")

        text_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/text.png"),
                                size=(24, 24))
        text_button = ctk.CTkButton(self, text="", image=text_image, width=30,
command=self.create_text)
        text_button.grid(row=2, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(text_button, msg="Text")

        pic_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/picture.png"),
                                size=(24, 24))
        pic_button = ctk.CTkButton(self, text="", image=pic_image, width=30,
command=self.create_picture)  # Changed
        pic_button.grid(row=2, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(pic_button, msg="Image")

        # Shape button handlers
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

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

    def create_tri(self):
        self.canvas.mouse.current_shape = "tri"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_text(self):
        self.canvas.mouse.current_shape = "text"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_picture(self):  # Added new method
        self.canvas.mouse.current_shape = "pic"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def hide_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = False
        self.canvas.redraw_shapes()
```

mouse.py modifications.

```
from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture  # Added Picture
from Shape_Lib import StraightLine, Connection

. . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "oval":
            self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
```

```
        elif self.current_shape == "tri":
            self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "text":
            self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "pic":  # Added
            self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)  # Added
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

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

. . .

    def select_hit_test(self, x, y):
        for s in self.canvas.shape_list:
            if isinstance(s, Text) or isinstance(s, Picture):  # Changed to
check for Picture
                if s.bbox[0] <= x <= s.bbox[2] and s.bbox[1] <= y <= s.bbox[3]:
                    self.selected_shape = s
                    s.is_selected = True
                    self.canvas.redraw_shapes()
                    return
            elif s.x1 <= x <= s.x2 and s.y1 <= y <= s.y2:
                self.selected_shape = s
                s.is_selected = True
                self.canvas.redraw_shapes()
                return

        # No shape hit - unselect all
        self.selected_shape = None
        self.unselect_all()

. . .
```

canvas.py modifications

```
import customtkinter as ctk
from tkinter import filedialog  # Added
from UI_LIb.mouse import Mouse
from Shape_Lib import Text


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

    def redraw_shapes(self):
        for s in self.shape_list:
            s.update()

    def edit_shape(self, _event):
        if self.mouse.selected_shape is not None:
            shape = self.mouse.selected_shape
            if self.gettags("current")[0] == "pics":  # Added
                filename = filedialog.askopenfilename(initialdir="../images",
title="select a file",  # Added
                                                      filetypes=(("png files",
"*.png"), ("all file", "*.*")))  # Added
                shape.filename = filename  # Added
                shape.update()  # Added
            elif isinstance(shape, Text):  # Changed
                dialog = ctk.CTkInputDialog(text="Enter new text", title="Edit
Text")
                shape.text = dialog.get_input()
                self.redraw_shapes()
```

Run the program, add two pictures of hamburgers (default), right click on one of the pictures and set it to a hot dog image. Verify that you can move, rotate, select, and connect to the image.

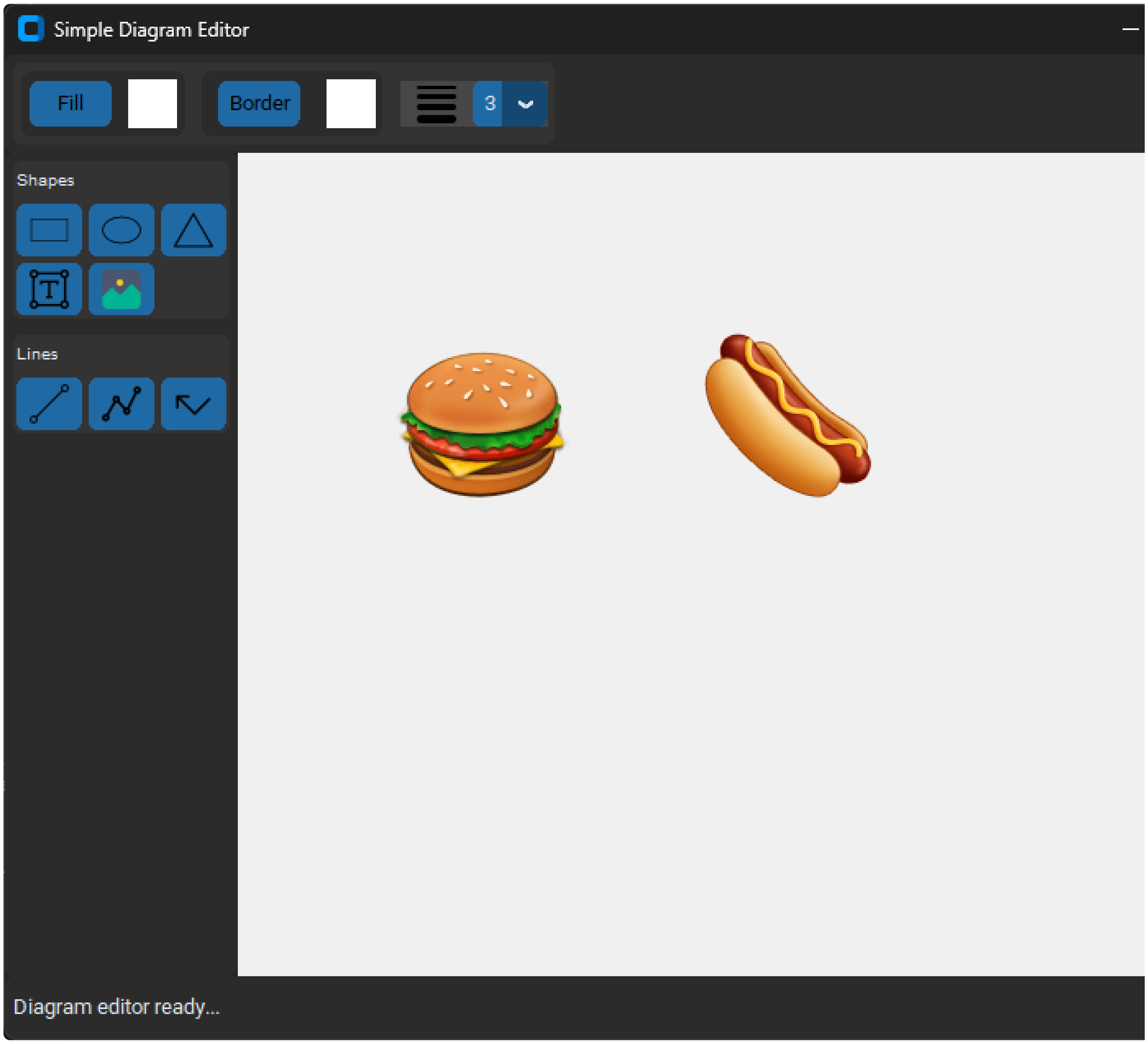


Diagram editor ready...

Segment Line Class

The segment line has three line segments. The `canvas.create_line()` method can draw lines with multiple segments. We will add a feature to change the starting direction to horizontal or vertical.
segment_line.py

```
from Shape_Lib.selector import Selector
from Shape_Lib.shape import Shape
```

```
class SegmentLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.fill_color = "black"
        self.border_width = 3
        self.line_direction = "horizontal"
        self.segment_list = []

        self.seg1_id, self.seg2_id, self.seg3_id = None, None, None

        self.s1_id, self.s2_id = None, None

        self.create_shape()
        self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        w = self.x2 - self.x1
        h = self.y2 - self.y1

        if self.line_direction == "horizontal":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1 + w /
2, self.y1,
                                              self.x1 + w / 2, self.y1, self.x1
+ w / 2, self.y2,
                                              self.x1 + w / 2, self.y2,
self.x2, self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width)
        elif self.line_direction == "vertical":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1,
self.y1 + h / 2,
                                              self.x1, self.y1 + h / 2,
self.x2, self.y1 + h / 2,
                                              self.x2, self.y1 + h / 2,
self.x2, self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width)

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

        if self.line_direction == "horizontal":
            self.canvas.coords(self.id,self.x1, self.y1, self.x1 + w / 2,
self.y1,
                                self.x1 + w / 2, self.y1, self.x1 + w / 2,
self.y2,
                                self.x1 + w / 2, self.y2, self.x2, self.y2)
        elif self.line_direction == "vertical":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y1 + h
/ 2,
                                self.x1, self.y1 + h / 2, self.x2, self.y1 + h /
2,
                                self.x2, self.y1 + h / 2, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)
```

line_button_frame.py modifications.

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image


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

        # Add left frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Lines", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        straight_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                size=(24, 24))
        straight_button = ctk.CTkButton(self, text="", image=straight_image,
width=30,
                                        command=self.create_straight_line)
        straight_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(straight_button, msg="Straight Line")
        segment_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                    dark_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                    size=(24, 24))
        segment_button = ctk.CTkButton(self, text="", image=segment_image,
width=30,  # Changed
                                            command=self.create_segment_line)    #
Changed
        segment_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(segment_button, msg="Segment Line")

        elbow_image = ctk.CTkImage(light_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
                                    dark_image=Image.open
("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
                                    size=(24, 24))
        elbow_button = ctk.CTkButton(self, text="", image=elbow_image,
width=30)
        elbow_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)
        ToolTip(elbow_button, msg="Elbow Line")

    # Shape button handlers
    def create_straight_line(self):
        self.canvas.mouse.current_shape = "straight"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_segment_line(self):  # Added new method
        self.canvas.mouse.current_shape = "segment"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def show_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = True
        self.canvas.redraw_shapes()
```

Shape_Lib/`__init__.py` modifications

```
# Import shapes
from Shape_Lib.rectangle import Rectangle
from Shape_Lib.oval import Oval
from Shape_Lib.triangle import Triangle
from Shape_Lib.text import Text
from Shape_Lib.picture import Picture

# Import lines
from Shape_Lib.straight_line import StraightLine
from Shape_Lib.segment_line import SegmentLine  # Added
from Shape_Lib.connection import Connection
```

mouse.py modifications

```
from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture
from Shape_Lib import StraightLine, SegmentLine, Connection  # Changed

. . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "oval":
            self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "tri":
            self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "text":
            self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "pic":
            self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
```

```
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
        elif self.current_shape == "segment":  # Added
            self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)  # Added
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)  # Added

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


. . .

    def draw_left_up(self, event):
        if isinstance(self.current_shape_obj, StraightLine) or
isinstance(self.current_shape_obj, SegmentLine):  # Changed
            self.select_connector(self.current_shape_obj, "end", event.x,
event.y)

        self.current_shape = None
        self.current_shape_obj = None
        self.unbind_mouse_events()
        self.move_bind_mouse_events()


. . .

   def select_connector(self, line_obj, line_end, x, y):
        for shape in self.canvas.shape_list:
            if not isinstance(shape, StraightLine) or not isinstance(shape,
SegmentLine):  # Changed
                conn = shape.check_connector_hit(x, y)
                if conn:
                    if line_end == "begin":
                        line_obj.x1, line_obj.y1 = conn.x, conn.y
                    elif line_end == "end":
                        line_obj.x2, line_obj.y2 = conn.x, conn.y
                    a_conn = Connection(conn, line_obj, line_end)
                    shape.line_list.append(a_conn)
                    self.canvas.redraw_shapes()
                    return
```

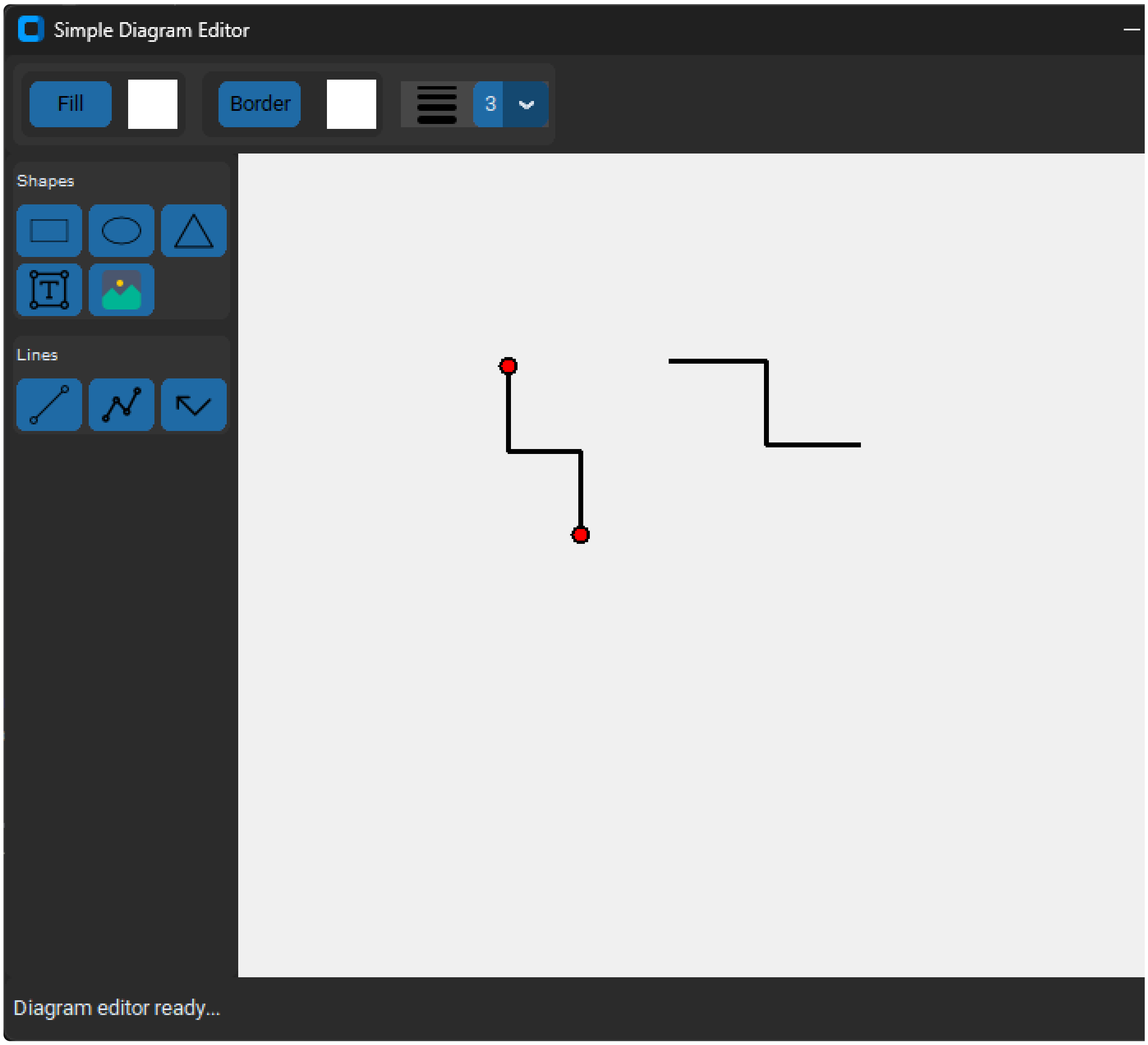
keyboard.py modifications.

```python
class Keyboard:
    def __init__(self, parent, canvas):
        """Class to manage keyboard events"""
        self.parent = parent
        self.canvas = canvas
        self.selected_shape = None

        # Declare keyboard bindings
        self.parent.bind('<r>', self.rotate_shape)
        self.parent.bind('<h>', self.set_horizontal)  # Added
        self.parent.bind('<v>', self.set_vertical)  # Added

    def rotate_shape(self, _event):
        for s in self.canvas.shape_list:
            if s.is_selected:
                s.rotate()
        self.canvas.redraw_shapes()

    def set_horizontal(self, _event):  # Added
        self.canvas.line_direction = "horizontal"  # Added

    def set_vertical(self, _event):  # Added
        self.canvas.line_direction = "vertical"  # Added
```

canvas.py modifications.

```python
. . .

class Canvas(ctk.CTkCanvas):
    def __init__(self, master):
        super().__init__(master)
        self.shape_list = []
        self.mouse = Mouse(self)
        self.line_direction = "horizontal"  # Added

. . .
```

Run the program, press the 'v' key, draw a segment line, press the 'h' key, draw another segment line.



Drawing segment lines is not as easy as it looks but we successfully programmed the new class.

Elbow Line Class

The Elbow Line Class is a simplified segment line with two segments in an L-shape or elbow-shape.
elbow_line.py

```
from Shape_Lib.selector import Selector
from Shape_Lib.shape import Shape
```

```
class ElbowLine(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.fill_color = "black"
        self.border_width = 3
        self.line_direction = self.canvas.line_direction
        self.segment_list = []

        self.seg1_id, self.seg2_id, self.seg3_id = None, None, None

        self.s1_id, self.s2_id = None, None

        self.create_shape()
        self.create_selectors()

    def create_shape(self):
        """Create the shape here"""
        if self.line_direction == "horizontal":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x2,
self.y1,
                                              self.x2, self.y1, self.x2,
self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width)
        elif self.line_direction == "vertical":
            self.id = self.canvas.create_line(self.x1, self.y1, self.x1,
self.y2,
                                              self.x1, self.y2, self.x2,
self.y2,
                                              fill=self.fill_color,
                                              width=self.border_width)

    def update(self):
        if self.line_direction == "horizontal":
            self.canvas.coords(self.id, self.x1, self.y1, self.x2, self.y1,
                                        self.x2, self.y1, self.x2, self.y2)
        elif self.line_direction == "vertical":
            self.canvas.coords(self.id, self.x1, self.y1, self.x1, self.y2,
                                        self.x1, self.y2, self.x2, self.y2)
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

    def create_selectors(self):
        """Create four selectors at the corners here"""
        self.s1_id = Selector(self.canvas, "begin", self.x1, self.y1)
        self.s2_id = Selector(self.canvas, "end", self.x2, self.y2)

        self.sel_list = [self.s1_id, self.s2_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        self.s1_id.x, self.s1_id.y = self.x1, self.y1
        self.s1_id.update()

        self.s2_id.x, self.s2_id.y = self.x2, self.y2
        self.s2_id.update()

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "end":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            self.x2, self.y2 = x2, y2
        elif self.selector == "begin":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            self.x1, self.y1 = x1, y1

    def check_selector_hit(self, x, y):
        for sel in self.sel_list:
            if sel.selector_hit_test(x, y):
                return sel
        return None

    def __repr__(self):
        return "Line: x1, y1: " + str(self.x1) + ", " + str(self.y1) + " x2,
y2: " + str(self.x2) + ", " + str(self.y2)
```

line_button_frame.py

```
import customtkinter as ctk
from tktooltip import ToolTip
from PIL import Image


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

        # Add left frame widgets here
        frame_name_label = ctk.CTkLabel(self, text="Lines", font=("Helvetica",
10), height=20)
        frame_name_label.grid(row=0, column=0, columnspan=3, sticky=ctk.W,
padx=2, pady=2)

        straight_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                        dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/line.png"),
                                        size=(24, 24))
        straight_button = ctk.CTkButton(self, text="", image=straight_image,
width=30,
                                                command=self.create_straight_line)
        straight_button.grid(row=1, column=0, sticky=ctk.W, padx=2, pady=2)
        ToolTip(straight_button, msg="Straight Line")

        segment_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                        dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/segment_line.png"),
                                        size=(24, 24))
        segment_button = ctk.CTkButton(self, text="", image=segment_image,
width=30,
                                                command=self.create_segment_line)
        segment_button.grid(row=1, column=1, sticky=ctk.W, padx=2, pady=2)
        ToolTip(segment_button, msg="Segment Line")

        elbow_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
                                dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/elbow.png"),
                                size=(24, 24))
        elbow_button = ctk.CTkButton(self, text="", image=elbow_image,
width=30,  # Changed
                                        command=self.create_elbow_line)  #
Changed
        elbow_button.grid(row=1, column=2, sticky=ctk.W, padx=2, pady=2)
        ToolTip(elbow_button, msg="Elbow Line")

    # Shape button handlers
    def create_straight_line(self):
        self.canvas.mouse.current_shape = "straight"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_segment_line(self):
        self.canvas.mouse.current_shape = "segment"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def create_elbow_line(self):  # Added new method
        self.canvas.mouse.current_shape = "elbow"
        self.show_connectors()
        self.canvas.mouse.draw_bind_mouse_events()

    def show_connectors(self):
        for s in self.canvas.shape_list:
            s.is_drawing = True
        self.canvas.redraw_shapes()
```

`UI_Lib/__init__.py`

```
# Import shapes
from Shape_Lib.rectangle import Rectangle
from Shape_Lib.oval import Oval
from Shape_Lib.triangle import Triangle
from Shape_Lib.text import Text
from Shape_Lib.picture import Picture

# Import lines
from Shape_Lib.straight_line import StraightLine
from Shape_Lib.segment_line import SegmentLine
from Shape_Lib.elbow_line import ElbowLine
from Shape_Lib.connection import Connection
```

mouse.py modifications

```
from Helper_Lib.point import Point
from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture
from Shape_Lib import StraightLine, SegmentLine, ElbowLine, Connection  #
Changed

. . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "oval":
            self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "tri":
            self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "text":
            self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "pic":
            self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
        elif self.current_shape == "segment":
            self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
```

```
        elif self.current_shape == "elbow":  # Added
            self.current_shape_obj = ElbowLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)  # Added
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

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

. . .

    def draw_left_up(self, event):
        if (isinstance(self.current_shape_obj, StraightLine) or
isinstance(self.current_shape_obj, SegmentLine)  # Changed
                or isinstance(self.current_shape_obj, ElbowLine)):  # Changed
            self.select_connector(self.current_shape_obj, "end", event.x,
event.y)

        self.current_shape = None
        self.current_shape_obj = None
        self.unbind_mouse_events()
        self.move_bind_mouse_events()

. . .

    def select_connector(self, line_obj, line_end, x, y):
        for shape in self.canvas.shape_list:
            if not isinstance(shape, StraightLine) or not isinstance(shape,
SegmentLine)\  # Changed
                    or not isinstance(shape, ElbowLine):  # Changed
                conn = shape.check_connector_hit(x, y)
                if conn:
                    if line_end == "begin":
                        line_obj.x1, line_obj.y1 = conn.x, conn.y
                    elif line_end == "end":
                        line_obj.x2, line_obj.y2 = conn.x, conn.y
                    a_conn = Connection(conn, line_obj, line_end)
                    shape.line_list.append(a_conn)
                    self.canvas.redraw_shapes()
                    return
```

Run the program, draw a horizontal (default) elbow line, press the 'v' key, draw a vertical elbow line.

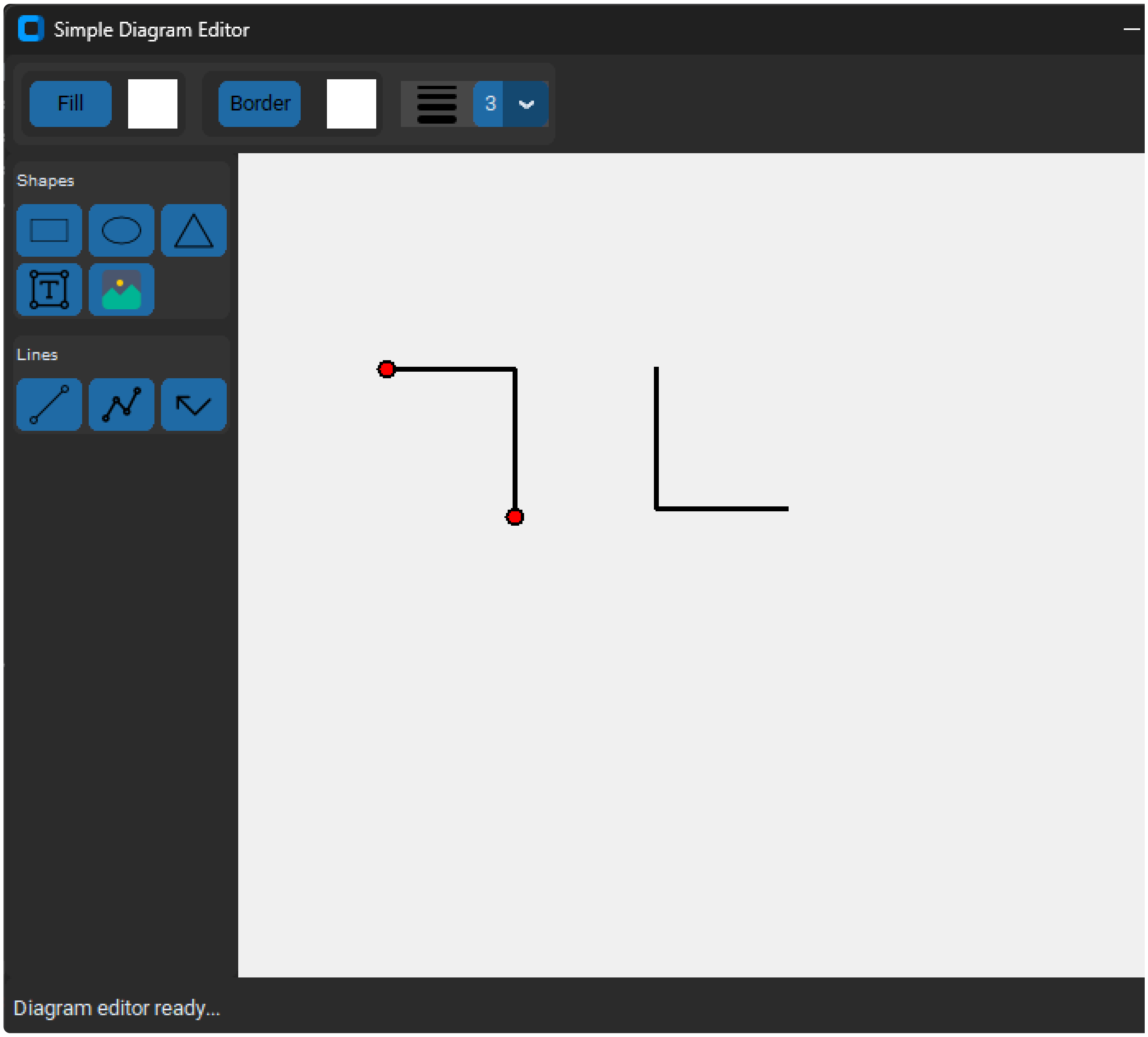


Diagram editor ready...

Grid Class

The Grid Class draws a background grid on the canvas at grid spacing intervals such as 10 px or 20 px. The `grid_snap` variable will set the snap size for shape draw, move, and resize. It also provide a Snap to Grid method that adjusts the provided x, y coordinates to the nearest snap point using the round() method.

grid.py

```
class Grid:
    def __init__(self, canvas, grid_size):
        self.canvas = canvas
        self.grid_size = grid_size
        self.grid_visible = True
        self.dash_list = None

        self.grid_snap = self.grid_size

    def draw(self):
        if self.grid_visible:
            w = self.canvas.winfo_width()  # Get current width of canvas
            h = self.canvas.winfo_height()  # Get current height of canvas

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

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

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

In the Main Application program we will add a binding to the `<configure>` event to detect a window resize. The window resize event handler will redraw the grid and all shapes.
simple_diagram_editor.py

```
import customtkinter as ctk
from UI_LIb import Canvas, TopFrame, LeftFrame, BottomFrame, Keyboard


class SimpleDiagramEditorApp(ctk.CTk):
    def __init__(self):
        super().__init__()
        self.geometry("800x600x100x100")  # w, h, x, y
        self.title("Simple Diagram Editor")

        self.canvas = Canvas(self)
        self.top_frame = TopFrame(self, self.canvas)
        self.left_frame = LeftFrame(self, self.canvas)
        self.bottom_frame = BottomFrame(self)

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

        self.keyboard = Keyboard(self, self.canvas)
        self.bind("<Configure>", self.on_window_resize)  # Added

    def on_window_resize(self, _event):  # Added
        self.canvas.redraw_shapes()  # Added


if __name__ == "__main__":
    app = SimpleDiagramEditorApp()
    app.mainloop()
```

Next we need to modify the Canvas Class redraw shapes method to draw the grid and lower its level (z-order) so it is always drawn in the background. We will also instantiate the Grid object in the Canvas Class initializer so it is drawn when the canvas is created.
canvas.py

```
import customtkinter as ctk
from tkinter import filedialog
from UI_LIb.mouse import Mouse
from Shape_Lib import Text
from Shape_Lib.grid import Grid  # Added


class Canvas(ctk.CTkCanvas):
    def __init__(self, master):
        super().__init__(master)
        self.shape_list = []
        self.mouse = Mouse(self)
        self.line_direction = "horizontal"
        self.grid_size = 10  # Added
        self.grid = Grid(self, self.grid_size)  # Added
        self.grid.draw()  # Added

    def redraw_shapes(self):
        self.delete('grid_line')  # Added
        self.grid.draw()  # Added
        self.tag_lower('grid_line')  # Added
        for s in self.shape_list:
            s.update()

    def edit_shape(self, _event):
        if self.mouse.selected_shape is not None:
            shape = self.mouse.selected_shape
            if self.gettags("current")[0] == "pics":
                filename = filedialog.askopenfilename(initialdir="../images",
title="select a file",
                                                      filetypes=(("png files",
"*.png"), ("all file", "*.*")))
                shape.filename = filename
                shape.update()
            elif isinstance(shape, Text):
                dialog = ctk.CTkInputDialog(text="Enter new text", title="Edit
Text")
                shape.text = dialog.get_input()
                self.redraw_shapes()
```

How do we snap shapes to the grid? We need to modify the Mouse Class to whenever coordinates are changed in the draw, move, and resize mouse handlers.
mouse.py

```
. . .

    def move_left_down(self, event):
        x, y = event.x, event.y
        self.select_hit_test(x, y)
        if self.selected_shape:
            # print("Shape found: ", self.selected_shape)
            sel = self.selected_shape.check_selector_hit(x, y)
            if sel:
                self.selected_shape.selector = sel.name
                self.unbind_mouse_events()
                self.bind_resize_mouse_events()
                self.resize_left_down(event)
                return
            else:
                x1, y1 = self.selected_shape.x1, self.selected_shape.y1
                x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)  # Added
                x2, y2 = self.selected_shape.x2, self.selected_shape.y2
                x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)  # Added
                self.offset1.x = x - x1
                self.offset1.y = y - y1
                self.offset2.x = x - x2
                self.offset2.y = y - y2

    def move_left_drag(self, event):
        if self.selected_shape:
            x = event.x - self.offset1.x
            y = event.y - self.offset1.y
            x, y = self.canvas.grid.snap_to_grid(x, y)  # Added
            self.selected_shape.x1, self.selected_shape.y1 = x, y
            x = event.x - self.offset2.x
            y = event.y - self.offset2.y
            x, y = self.canvas.grid.snap_to_grid(x, y)  # Added
            self.selected_shape.x2, self.selected_shape.y2 = x, y
            self.canvas.redraw_shapes()

. . .

    def draw_left_down(self, event):
        self.unselect_all()
        self.start.x = event.x
        self.start.y = event.y
        self.start.x, self.start.y =
self.canvas.grid.snap_to_grid(self.start.x, self.start.y)  # Added

        if self.current_shape == "rectangle":
            self.current_shape_obj = Rectangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "oval":
            self.current_shape_obj = Oval(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "tri":
            self.current_shape_obj = Triangle(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "text":
            self.current_shape_obj = Text(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "pic":
            self.current_shape_obj = Picture(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
        elif self.current_shape == "straight":
            self.current_shape_obj = StraightLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
        elif self.current_shape == "segment":
            self.current_shape_obj = SegmentLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)
        elif self.current_shape == "elbow":
            self.current_shape_obj = ElbowLine(self.canvas, self.start.x,
self.start.y, self.start.x, self.start.y)
            self.select_connector(self.current_shape_obj, "begin",
self.start.x, self.start.y)

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

    def draw_left_drag(self, event):
        if self.current_shape_obj:
            shape = self.current_shape_obj
            x, y = event.x, event.y
            x, y = self.canvas.grid.snap_to_grid(x, y)  # Added
            shape.x1, shape.y1 = self.start.x, self.start.y
            shape.x2, shape.y2 = x, y
            self.canvas.redraw_shapes()

. . .

    def resize_left_down(self, event):
        if self.selected_shape:
            shape = self.selected_shape
            x1, y1 = shape.x1, shape.y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)  # Added
            x2, y2 = shape.x2, shape.y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)  # Added
            self.offset1.x = event.x - x1
            self.offset1.y = event.y - y1
            self.offset2.x = event.x - x2
            self.offset2.y = event.y - y2

. . .
```

Next we need to modify the resize() method in the Rectangle, Oval, and Triangle classes.
rectangle.py

```
. . .

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)  # Added
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            x2, y1 = self.canvas.grid.snap_to_grid(x2, y1)  # Added
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)  # Added
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)  # Added
            self.x1, self.y2 = x1, y2

 . . .
```

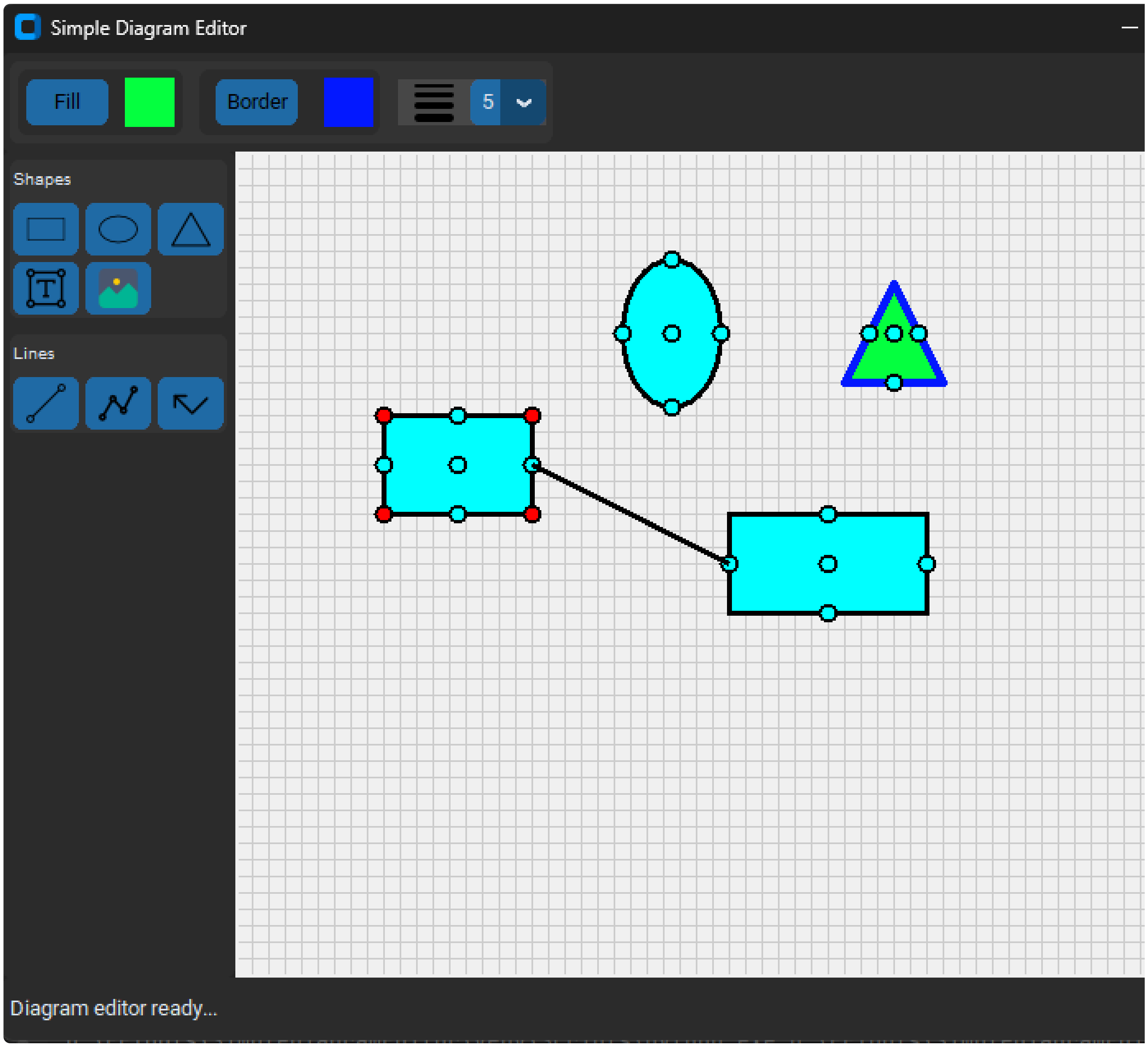
oval.py

```
. . .

    def resize(self, offsets, event):
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)  # Added
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y1 = event.y - offset_y1
            x2, y1 = self.canvas.grid.snap_to_grid(x2, y1)  # Added
            self.x2, self.y1 = x2, y1
        elif self.selector == "s3":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)  # Added
            self.x2, self.y2 = x2, y2
        elif self.selector == "s4":
            x1 = event.x - offset_x1
            y2 = event.y - offset_y2
            x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)  # Added
            self.x1, self.y2 = x1, y2

 . . .
```

Oops, we forgot to add a Triangle Class resize and rotate methods. Lets add them now and with `snap_to_grid()` method calls where needed.
triangle.py

```
from Helper_Lib import Point
from Shape_Lib.shape import Shape
from Shape_Lib.selector import Selector
from Shape_Lib.connector import Connector


class Triangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2):
        super().__init__(canvas, x1, y1, x2, y2)
        self.points = []
        self.type = "triangle"
        self.angle = 0

        self.s1_id, self.s2_id, self.s3_id = None, None, None
        self.c1_id, self.c2_id, self.c3_id, self.c4_id = None, None, None, None

        self.create_shape()
        self.create_selectors()
        self.create_connectors()

    def create_shape(self):
        """Create the shape once!"""
        w = self.x2 - self.x1
        self.points = [self.x1 + w / 2, self.y1, self.x2, self.y2, self.x1,
self.y2]
        self.id = self.canvas.create_polygon(self.points, fill=self.fill_color,
outline=self.border_color,
                                             width=self.border_width)
    def update(self):
        w = self.x2 - self.x1
        self.rotation_points()
        self.canvas.coords(self.id, self.points[0], self.points[1],
self.points[2], self.points[3],
                           self.points[4], self.points[5])
        self.canvas.itemconfig(self.id, fill=self.fill_color)
        self.canvas.itemconfig(self.id, outline=self.border_color)
        self.canvas.itemconfig(self.id, width=self.border_width)

        self.update_selectors()
        if self.is_selected:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='normal')
        else:
            for s in self.sel_list:
                self.canvas.itemconfig(s.id, state='hidden')

        self.update_connectors()
        if self.is_drawing:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='normal')
        else:
            for c in self.conn_list:
                self.canvas.itemconfig(c.id, state='hidden')

        self.move_connected_lines()

    def create_selectors(self):
        """Create four selectors at the corners here"""
        w = self.x2 - self.x1
        self.s1_id = Selector(self.canvas, "s1", self.x1 + w / 2, self.y1)
        self.s2_id = Selector(self.canvas, "s2", self.x2, self.y2)
        self.s3_id = Selector(self.canvas, "s3", self.x1, self.y2)

        self.sel_list = [self.s1_id, self.s2_id, self.s3_id]
        for s in self.sel_list:
            self.canvas.itemconfig(s.id, state='hidden')

    def update_selectors(self):
        """Update the position of all selectors here"""
        x1, y1, x2, y2, x3, y3 = self.points
        w = x2 - x1
        self.s1_id.x, self.s1_id.y = x1, y1
        self.s1_id.update()
        self.s2_id.x, self.s2_id.y = x2, y2
        self.s2_id.update()

        self.s3_id.x, self.s3_id.y = x3, y3
        self.s3_id.update()

    def create_connectors(self):
        """Create connectors here"""
        # Calculate the shape geometry
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        center = Point(self.x1+w/2, self.y1+h/2)

        self.c1_id = Connector(self.canvas, "c1", center.x, center.y)
# Shape Center
        self.c2_id = Connector(self.canvas, "c2", center.x, center.y - h / 2)
# Right Center
        self.c3_id = Connector(self.canvas, "c3", center.x + w / 4, center.y)
# Left Center
        self.c4_id = Connector(self.canvas, "c4", center.x, center.y + h / 2)
# Bottom Center

        self.conn_list = [self.c1_id, self.c2_id, self.c3_id, self.c4_id]
        for c in self.conn_list:
            self.canvas.itemconfig(c.id, state='hidden')

    def update_connectors(self):  # Modified method
        """Update the position of all connectors here"""
        x1, y1, x2, y2, x3, y3 = self.points


        if self.angle == 0 or self.angle == 180:
            w = x2 - x3
            h = y2 - y1
            center = Point(x1 + w / 2, y1 + h / 2)
            self.c1_id.x, self.c1_id.y = x1, y1+h/2
            self.c1_id.update()
            self.c2_id.x, self.c2_id.y = x1 - w/4, center.y
            self.c2_id.update()
            self.c3_id.x, self.c3_id.y = x1 + w/4, center.y
            self.c3_id.update()
            self.c4_id.x, self.c4_id.y = x1, y2
            self.c4_id.update()
        elif self.angle == 90 or self.angle == 270:
            w = x2 - x1
            h = y2 - y3
            center = Point(x1 + w / 2, y1)
            self.c1_id.x, self.c1_id.y = x1 + w/2, y1
            self.c1_id.update()
            self.c2_id.x, self.c2_id.y = center.x, center.y - h/4
            self.c2_id.update()
            self.c3_id.x, self.c3_id.y = center.x, center.y + h/4
            self.c3_id.update()
            self.c4_id.x, self.c4_id.y = x2, y1
            self.c4_id.update()

    def rotate(self):  # Added method
        # Calculate rotation angle
        self.angle += 90
        if self.angle > 270:
            self.angle = 0

    def rotation_points(self):  # Added method
        w = self.x2 - self.x1
        h = self.y2 - self.y1
        if self.angle == 0:
            self.points = [self.x1 + w / 2, self.y1, self.x2, self.y2, self.x1,
self.y2]
        elif self.angle == 90:
            self.points = [self.x1, self.y1 + h/2, self.x2, self.y1, self.x2,
self.y2]
        elif self.angle == 180:
            self.points = [self.x1 + w/2, self.y2, self.x1, self.y1, self.x2,
self.y1]
        elif self.angle == 270:
            self.points = [self.x2, self.y1 + h/2, self.x1, self.y2, self.x1,
self.y1]

    def resize(self, offsets, event):  # Added method
        offset_x1, offset_y1, offset_x2, offset_y2 = offsets
        if self.selector == "s1":
            x1 = event.x - offset_x1
            y1 = event.y - offset_y1
            x1, y1 = self.canvas.grid.snap_to_grid(x1, y1)
            self.x1, self.y1 = x1, y1
        elif self.selector == "s2":
            x2 = event.x - offset_x2
            y2 = event.y - offset_y2
            x2, y2 = self.canvas.grid.snap_to_grid(x2, y2)
            self.x2, self.y2 = x2, y2
        elif self.selector == "s3":
                x1 = event.x - offset_x1
                y2 = event.y - offset_y2
                x1, y2 = self.canvas.grid.snap_to_grid(x1, y2)
                self.x1, self.y2 = x1, y2

    def __repr__(self):
        return ("Triangle: x1, y1, x2, y2 = " + "(" + str(self.x1) + ", " +
str(self.y1) + ", " + str(self.y2) + ", " +
                str(self.y2) + ")\nfill color: " + self.fill_color +
"\nborder_color: " + self.border_color +
                "\nborder_width: " + str(self.border_width))
```

Run the program and draw rectangles, ovals, triangles and lines and verify that they snap to the grid spacing. Also verify that the triangle will rotate and resize.



Top Frame Class Update

Next we will add three menus to the Top Frame:

- File Menu
- Settings Menu
- Help Menu

File Menu Frame Class

file_menu_frame.py

```
import customtkinter as ctk
from tkinter import filedialog as fd
import json
from PIL import Image

from Shape_Lib import Rectangle, Oval, Triangle, Text, Picture
from Shape_Lib import StraightLine, SegmentLine, ElbowLine


class FileMenuFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.canvas = canvas
        self.obj_type_dict = {'rectangle': Rectangle,
                              'oval': Oval,
                              'triangle': Triangle,
                              'text': Text,
                              'straight': StraightLine,
                              'segment': SegmentLine,
                              'elbow': ElbowLine,
                              'picture': Picture}

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        new_btn = ctk.CTkButton(self.menu_frame, text="New", width=150,
command=self.new_diagram)
        new_btn.pack(pady=5)
        open_btn = ctk.CTkButton(self.menu_frame, text="Open", width=150,
command=self.load_diagram)
        open_btn.pack(pady=5)

        save_btn = ctk.CTkButton(self.menu_frame, text="Save", width=150,
command=self.save_diagram)
        save_btn.pack(pady=5)

        exit_btn = ctk.CTkButton(self.menu_frame, text="Exit", width=150,
command=window.destroy)
        exit_btn.pack(pady=5)

        my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/DiagramEditor/icons/hamburger_menu.png"),
                                dark_image=Image.open

("D:/EETools/DiagramEditor/icons/hamburger_menu.png"),
                                size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def new_diagram(self):
        self.save_diagram()
        self.canvas.delete("all")
        self.canvas.shape_list = []

    def load_diagram(self):
        try:
            filetypes = (('json files', '*.json'), ('All files', '*.*'))
            f = fd.askopenfilename(filetypes=filetypes, initialdir="./")
            with open(f) as file:
                obj_dict = json.load(file)
                for obj_type, attributes in obj_dict.items():
                    obj = self.obj_type_dict[obj_type.split()[0]](self.canvas,
attributes[0], attributes[1],

attributes[2], attributes[3])
                    self.canvas.shape_list.append(obj)
                self.canvas.redraw_shapes()

        except FileNotFoundError:
            with open('untitled.canvas', 'w') as file:
                pass
            self.canvas.shape_list = []

    def save_diagram(self):
        filetypes = (('json files', '*.json'), ('All files', '*.*'))
        f = fd.asksaveasfilename(filetypes=filetypes, initialdir="./")
        with open(f, 'w') as file:
            obj_dict = {f'{obj.type} {id}': (obj.x1, obj.y1, obj.x2, obj.y2)
for id, obj in
                        enumerate(self.canvas.shape_list)}
            json.dump(obj_dict, file)

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False
```

Settings Frame Class

settings_frame.py

```
import customtkinter as ctk
from PIL import Image


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

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        def grid_switch_event():
            if canvas.grid.grid_visible:
                canvas.grid.grid_visible = False
                self.canvas.mouse.move_snap = 1
                self.canvas.mouse.resize_snap = 1
                self.canvas.mouse.rotate_snap = 90
            else:
                canvas.grid.grid_visible = True
                self.canvas.mouse.move_snap = 10
                self.canvas.mouse.resize_snap = 10
                self.canvas.mouse.rotate_snap = 90
            self.canvas.redraw_shapes()

        switch_var = ctk.StringVar(value="on")
        switch = ctk.CTkSwitch(self.menu_frame, text="Grid",
command=grid_switch_event,
                                         variable=switch_var, onvalue="on",
offvalue="off")
        switch.pack(padx=5, pady=5)

        grid_size_label = ctk.CTkLabel(self.menu_frame, text="Grid Size", font=
("Helvetica", 10), height=20)
        grid_size_label.pack(padx=5, pady=5, anchor="w")

        def optionmenu_callback(choice):
            self.canvas.grid.grid_size = int(choice)
            self.canvas.redraw_shapes()

        optionmenu = ctk.CTkOptionMenu(self.menu_frame, values=["5", "10",
"20", "30", "40", "50"],
                                                 command=optionmenu_callback)
        optionmenu.pack(padx=5, pady=5)
        optionmenu.set("10")

        grid_snap_label = ctk.CTkLabel(self.menu_frame, text="Snap Size", font=
("Helvetica", 10), height=20)
        grid_snap_label.pack(padx=5, pady=5, anchor="w")

        def snap_option_callback(choice):
            if choice == "Grid Size":
                self.canvas.grid.grid_snap = self.canvas.grid.gri_size
            else:
                self.canvas.grid.grid_snap = int(choice)
            self.canvas.redraw_shapes()

        snap_option = ctk.CTkOptionMenu(self.menu_frame, values=["Grid Size",
"5", "10", "20", "30", "40", "50"],
                                                 command=snap_option_callback)
        snap_option.pack(padx=5, pady=5)
        snap_option.set("Grid Size")

        self.appearance_mode_label = ctk.CTkLabel(self.menu_frame,
text="Appearance Mode:", anchor="w")
        self.appearance_mode_label.pack(padx=5, pady=5)
        self.appearance_mode_optionemenu = ctk.CTkOptionMenu(self.menu_frame,
                                                                values=
["Light", "Dark", "System"],

command=self.change_appearance_mode_event)
        self.appearance_mode_optionemenu.pack(padx=5, pady=5)
        self.appearance_mode_optionemenu.set("Dark")

        my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/DiagramEditor/icons/settings.png"),
                                dark_image=Image.open

("D:/EETools/DiagramEditor/icons/settings.png"),
                                size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            self.menu_frame.place(x=15, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

    def change_appearance_mode_event(self, new_appearance_mode: str):
        ctk.set_appearance_mode(new_appearance_mode)
```

Help Frame Class

help_frame.py

```
import customtkinter as ctk
from tkinter import messagebox
from PIL import Image
```

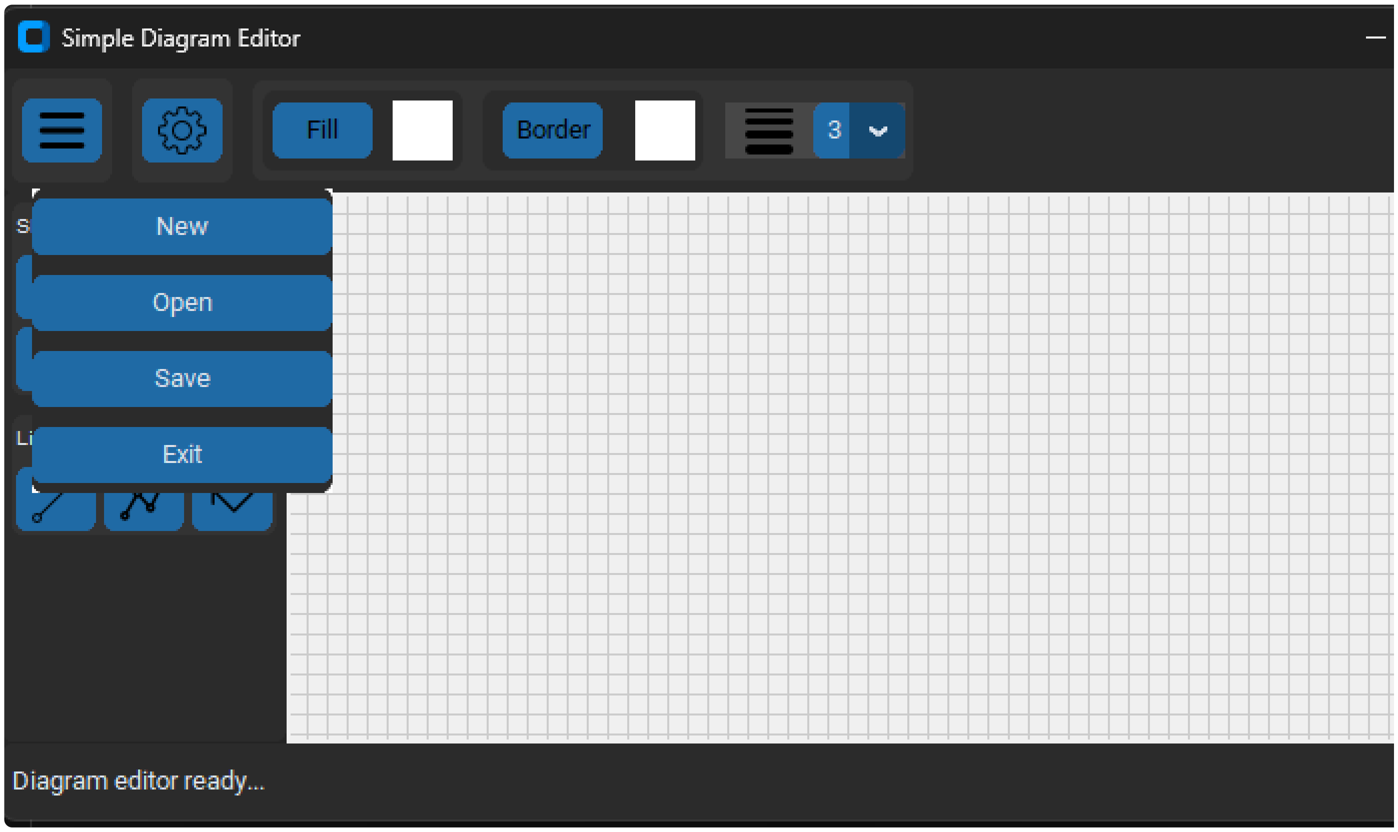
```
class HelpFrame(ctk.CTkFrame):
    def __init__(self, window, parent, canvas):
        super().__init__(parent)
        self.window = window
        self.canvas = canvas

        self.menu_on = False

        self.menu_frame = ctk.CTkFrame(window, height=100, bg_color="white")

        about_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/about.png"),
                                   dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/about.png"),
                                   size=(24, 24))

        about_button = ctk.CTkButton(self.menu_frame, text="About",
image=about_image, width=30,
                                        command=self.show_about_dialog)
        about_button.pack(side=ctk.TOP,padx=5, pady=5)

        my_image = ctk.CTkImage(light_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/help.png"),
                                   dark_image=Image.open

("D:/EETools/SimpleDiagramEditor/icons/help.png"),
                                   size=(24, 24))

        button = ctk.CTkButton(self, text="", image=my_image, width=30,
command=self.show_menu)
        button.pack(side=ctk.LEFT, padx=5, pady=10)

    def show_menu(self):
        if not self.menu_on:
            menu_pos_x = self.canvas.winfo_width()
            self.menu_frame.place(x=menu_pos_x + 50, y=60)
            self.menu_frame.tkraise()
            self.menu_on = True
        else:
            self.menu_frame.place_forget()
            self.menu_on = False

    def show_about_dialog(self):
        messagebox.showinfo("About Diagram Editor", "Diagram Editor v0.1\n" +
"Author: Rick A. Crist\n" + "2023")
```

Top Frame Class

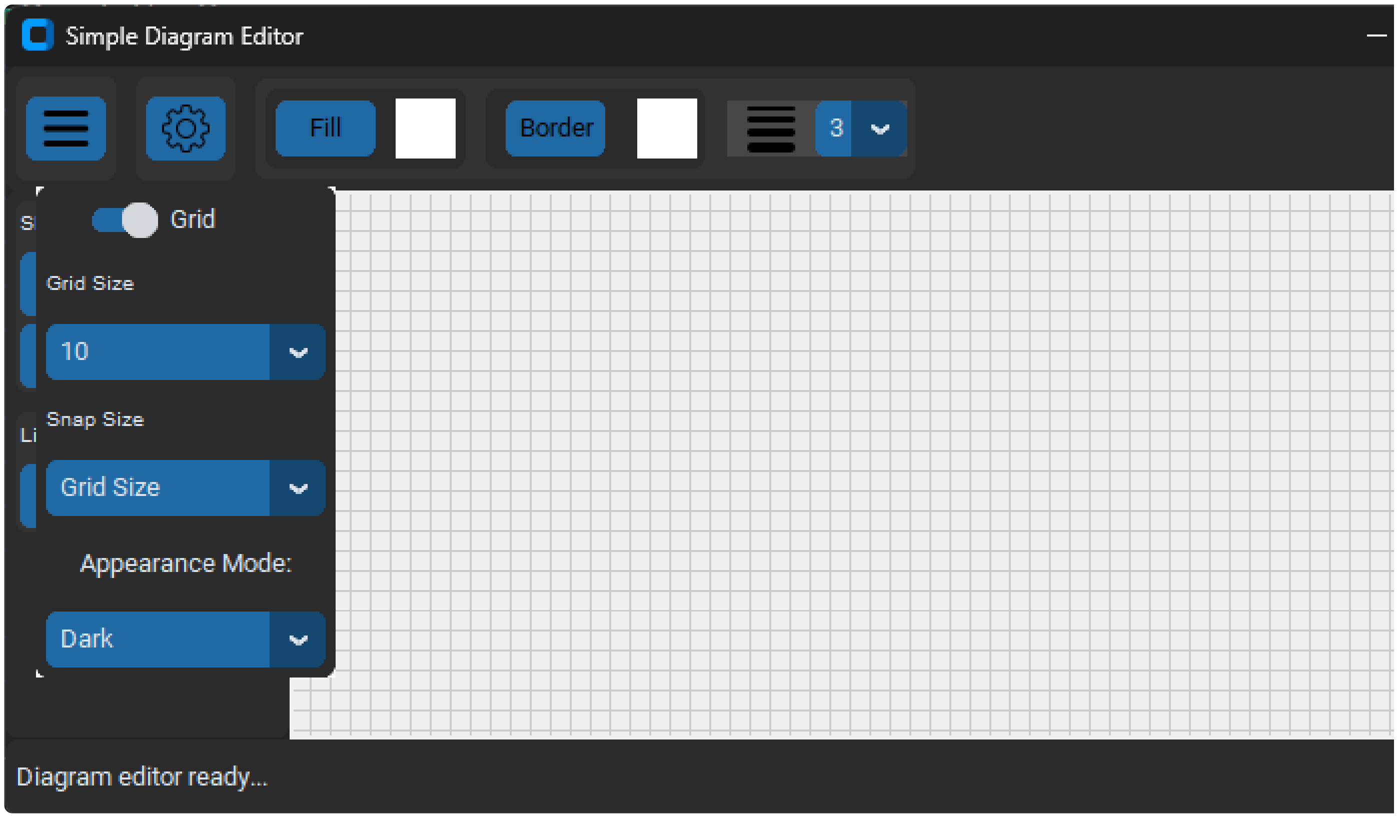
top_frame.py

```
import customtkinter as ctk
from UI_Lib.shape_appearance_frame import ShapeAppearanceFrame
from UI_Lib.file_menu_frame import FileMenuFrame
from UI_Lib.settings_frame import SettingsFrame
from UI_Lib.help_frame import HelpFrame


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

        # Add Top Frame widget here
        file_frame = FileMenuFrame(self.parent, self, self.canvas)
        file_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        settings_frame = SettingsFrame(self.parent, self, self.canvas)
        settings_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        shape_frame = ShapeAppearanceFrame(self, self.canvas)
        shape_frame.pack(side=ctk.LEFT, padx=5, pady=5)

        help_frame = HelpFrame(self.parent, self, self.canvas)
        help_frame.pack(side=ctk.RIGHT, padx=5, pady=5)
```

File Menu



Settings Menu



Help Menu

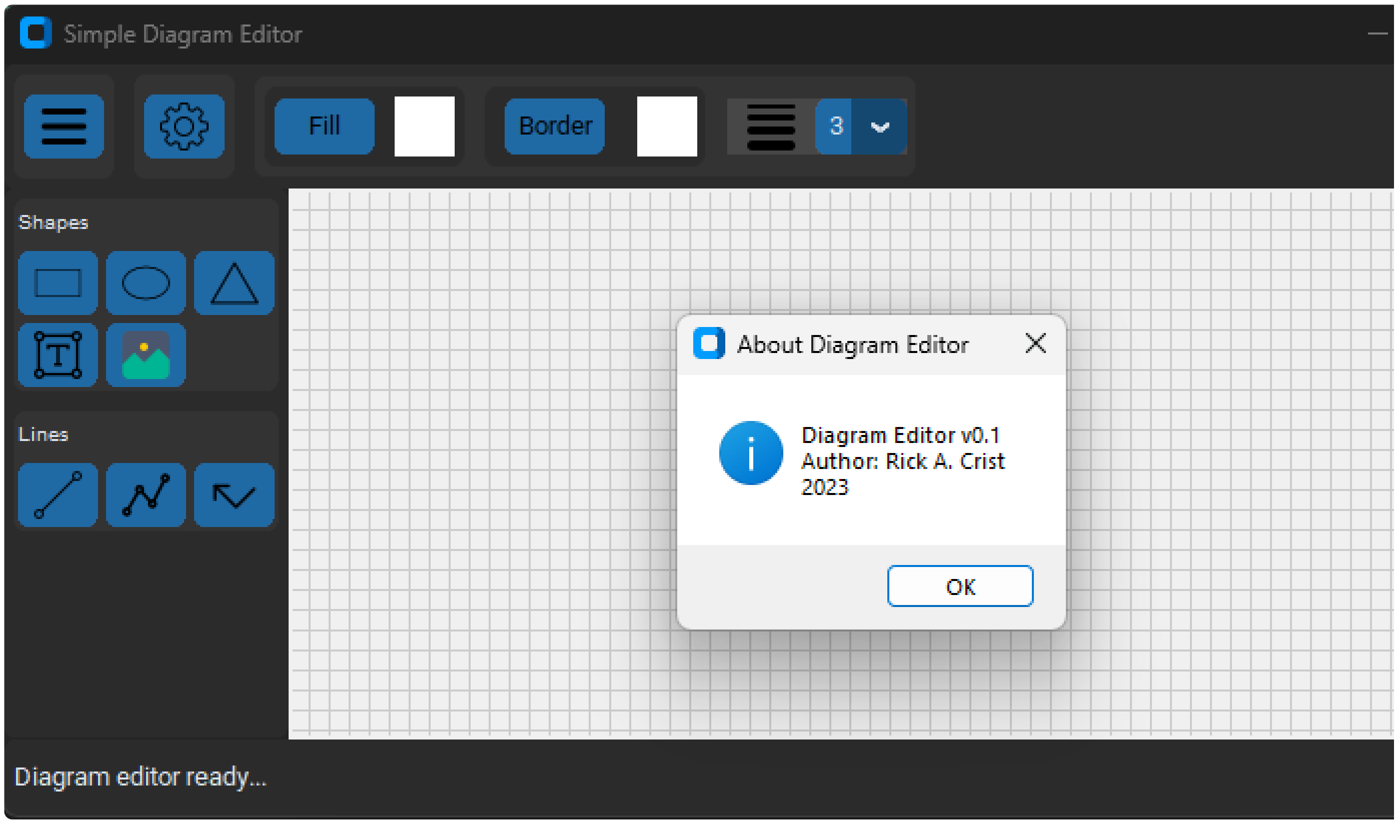


Diagram Editor v0.1
Author: Rick A. Crist
2023

Diagram editor ready...

Status Messages

Lets go through the button handlers and send a message to the status message bar at the bottom of the screen.
Here is a list of button handlers:

- Shape Button Frame - 5 buttons
- Line Button Frame - 3 buttons

First, we need to refactor the Bottom Frame Class to add an update() method and make the text an attribute of the class.
bottom_frame.py

```
import customtkinter as ctk


class BottomFrame(ctk.CTkFrame):
    def __init__(self, parent):
        super().__init__(parent)
        self.text = "Diagram editor ready..."  # Added
        self.message = ctk.CTkLabel(self, text=self.text)  # Changed
        self.message.pack(side=ctk.LEFT, padx=5, pady=5)

    def update(self):  # Added
        self.message.configure(text=self.text)  # Added
```

Here is an example from the Rectangle Button in the Shape Button Frame

```
    def create_rect(self):
        self.canvas.mouse.current_shape = "rectangle"
        self.hide_connectors()
        self.canvas.mouse.draw_bind_mouse_events()
        self.parent.parent.bottom_frame.text = "Drawing Rectangle"  # Added
        self.parent.parent.bottom_frame.update() # Added
```

Summary

This is the end of the Simple Diagram Editor application. We have come a long ways since starting this application development and due to its complexity, it is an advanced python project. It is also important for the future projects which all require a diagram capability to draw circuits and schematics.

You can model a circuit simulator as a Diagram Editor with a simulation capability. The next project is a Digital Circuit Simulator so lets proceed.