Chapter 3 - Shape Editor Project

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

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

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

Project Design, Features, and Specification

Required features:

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

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

Shape Creation and Drawing Algorithm

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

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

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

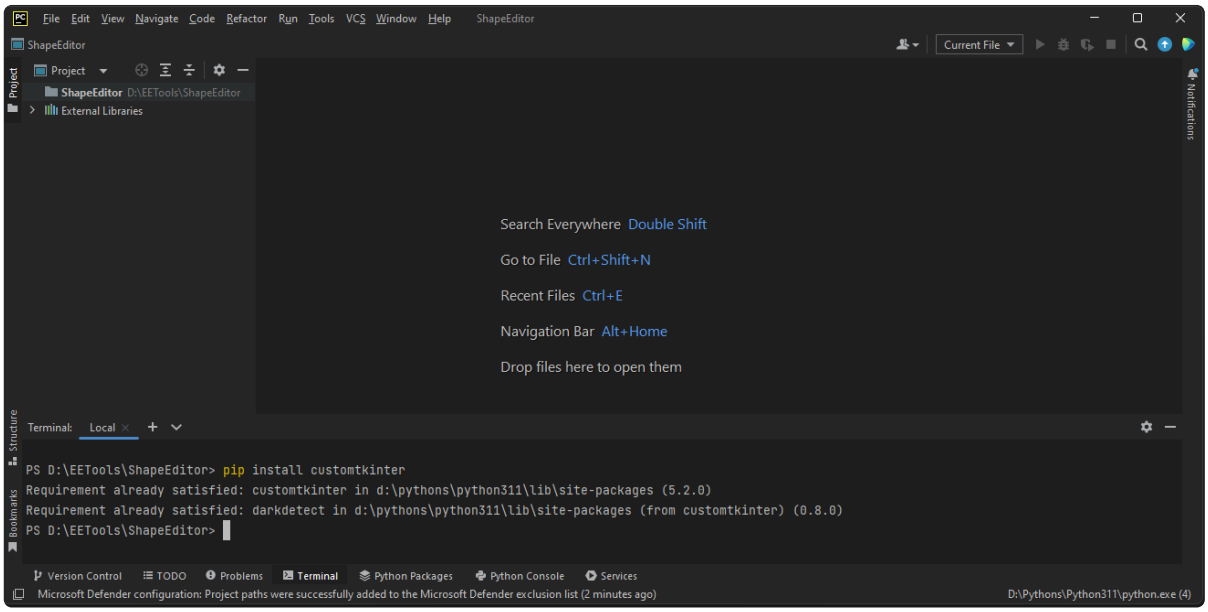
Project Setup

Enough talk, let's create a new project.

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

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

My Project Directory: D:/EETools/ShapeEditor



GUI Design and Layout

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

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

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

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

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

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

```
import customtkinter as ctk
```

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

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

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

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

shape_editor.py - complete program

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

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

import customtkinter as ctk


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

                # Add widgets to the app here


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

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

Draw a Red Rectangle

Canvas

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

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

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

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

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

Pack or Packer Layout Manger

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

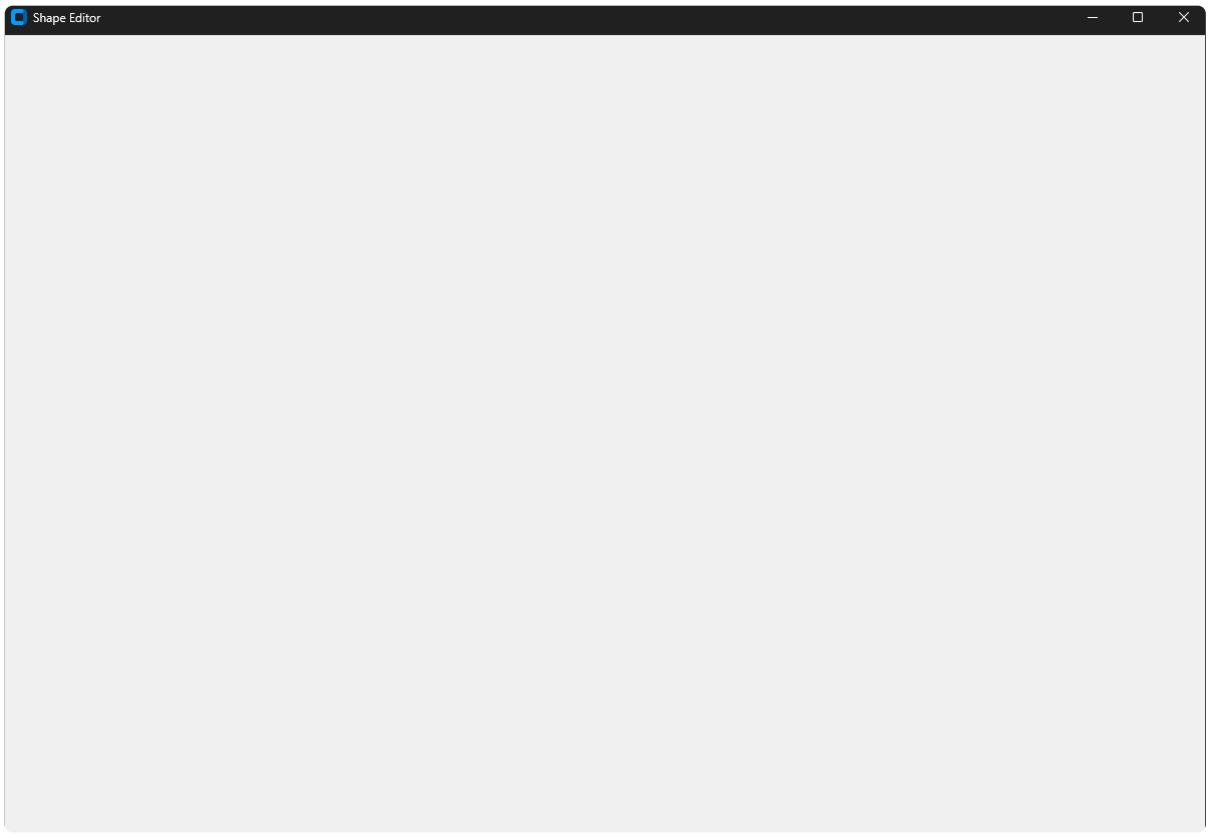
Add a pack statement to display the canvas.

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

pack options:

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

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



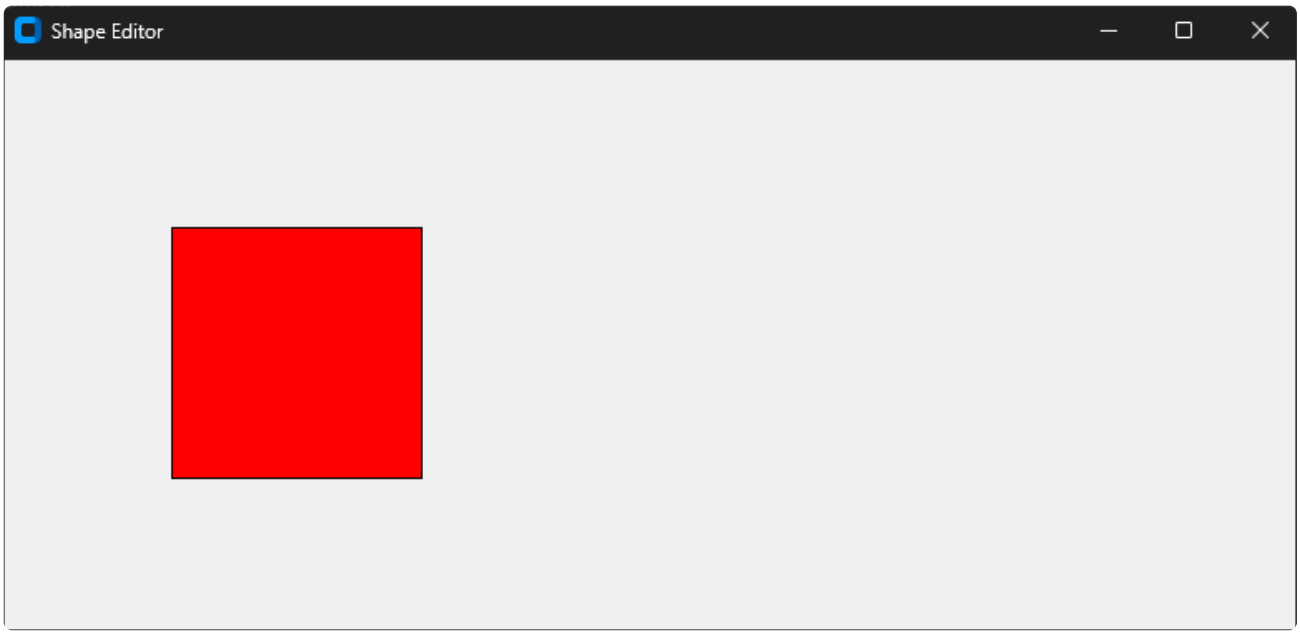
Drawing a Shape

The canvas class has a library for creation of shapes:

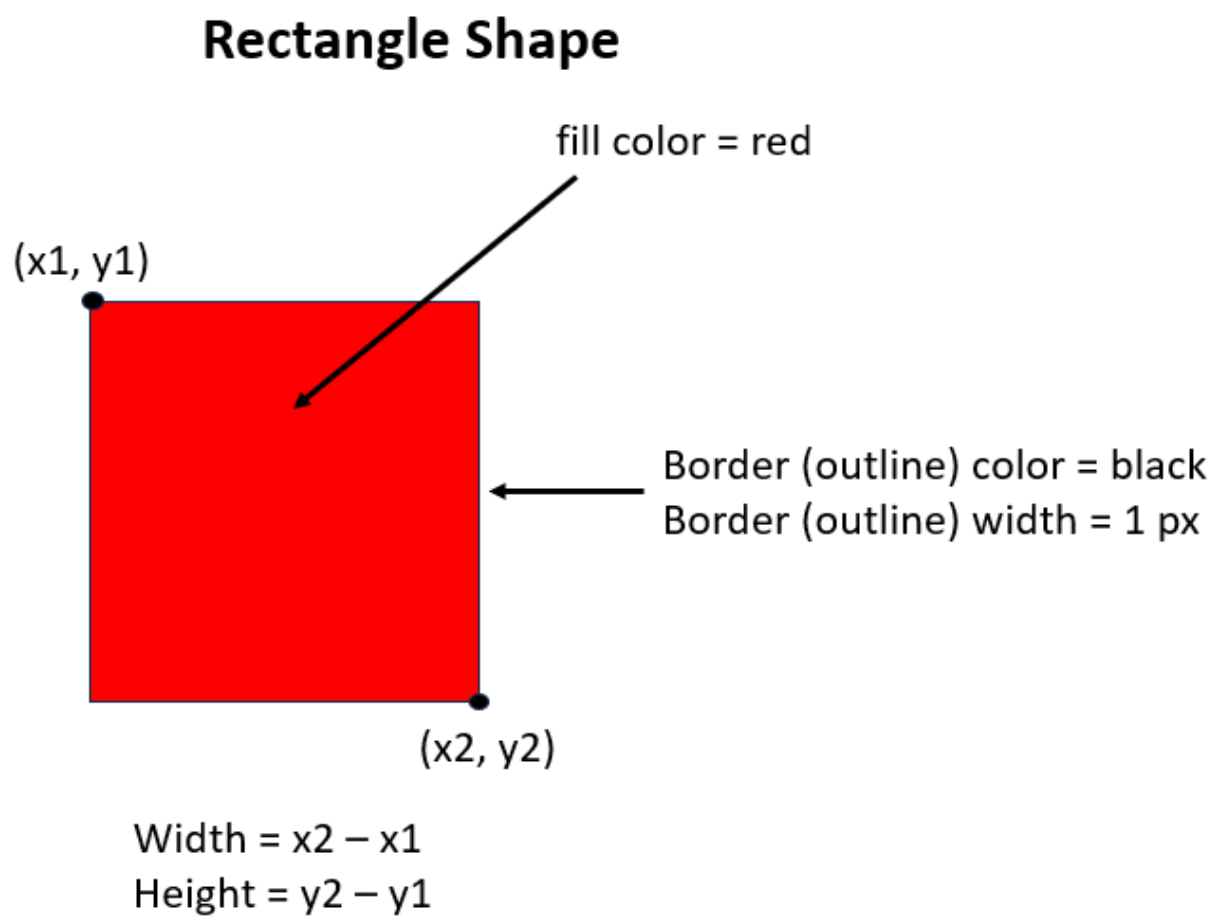
- `canvas.create_rectangle(x1, y1, x2, y2, option=value)`
- `canvas.create_oval(x1, y1, x2, y2, option=value)`
- `canvas.create_polygon(points_list, option=value)`
- `canvas.create_line(x1, y1, x2, y2, option=value)`
- `canvas.create_image(x, y, option=value)`
- `canvas.create_text(x, y, option=value)`

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

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

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



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



The rectangle is drawn from the coordinates representing the upper-right corner (x1, y1) to the coordinates representing the lower-left corner (x2, y2). Other geometrical parameters can be calculated from the "bounding box" coordinates as follows:

- Width = x2 - x1
- Height = y2 - y1

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

The appearance of the rectangle is defined by optional values.
`canvas.create_rectangle(x1, y1, x2, y2, fill=fill_color, outline=border_color, width=border_width)`

Three options provide control over the shape appearance:

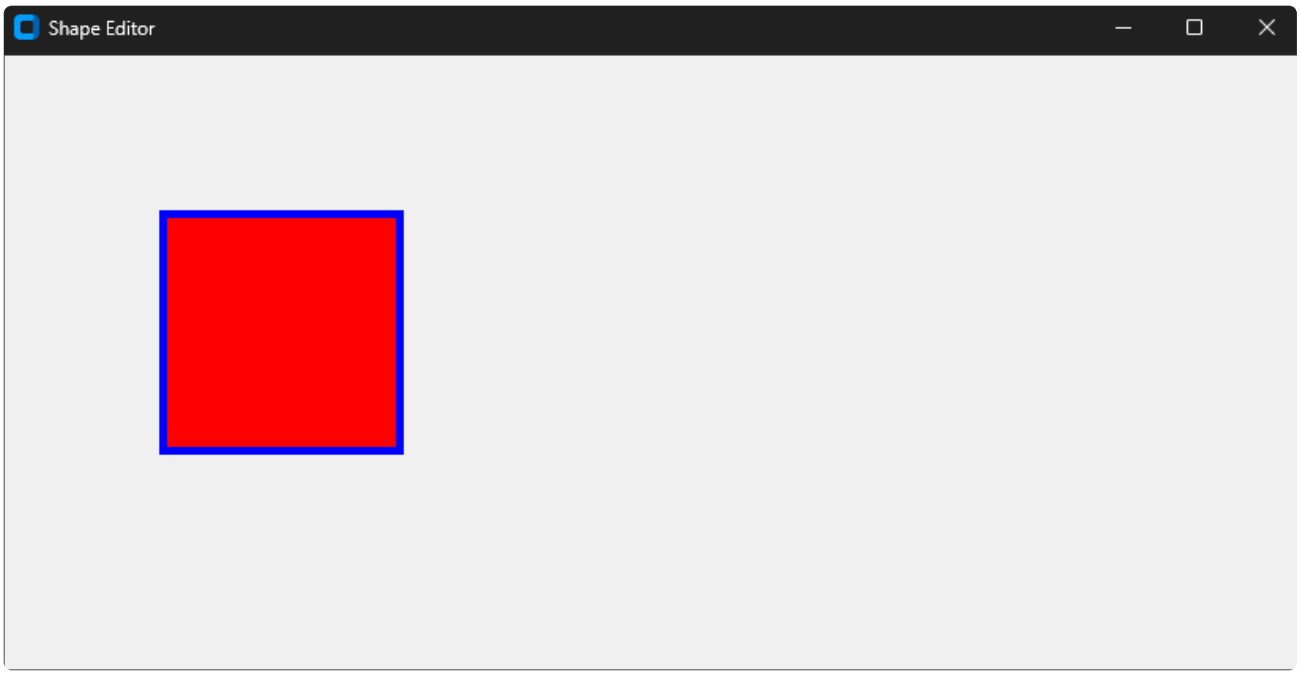
- `fill=fill_color`, example: fill="red"
- `outline=border_color`, example: outline="black"
- `width=border_width`, example: width=5

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

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

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

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



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

Frame

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

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

Top Frame

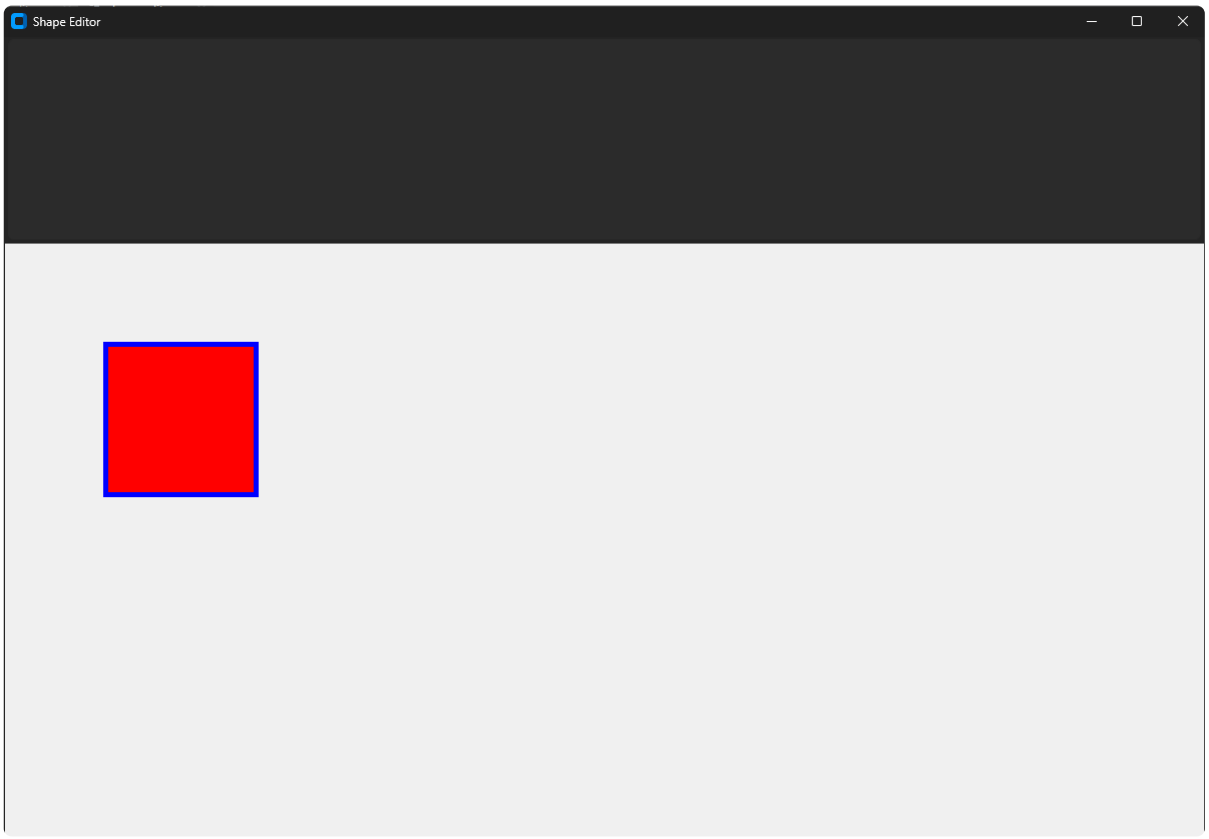
Add a frame called `top_frame` to the application.

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

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

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

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



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

Shape Appearance Controls

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

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

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

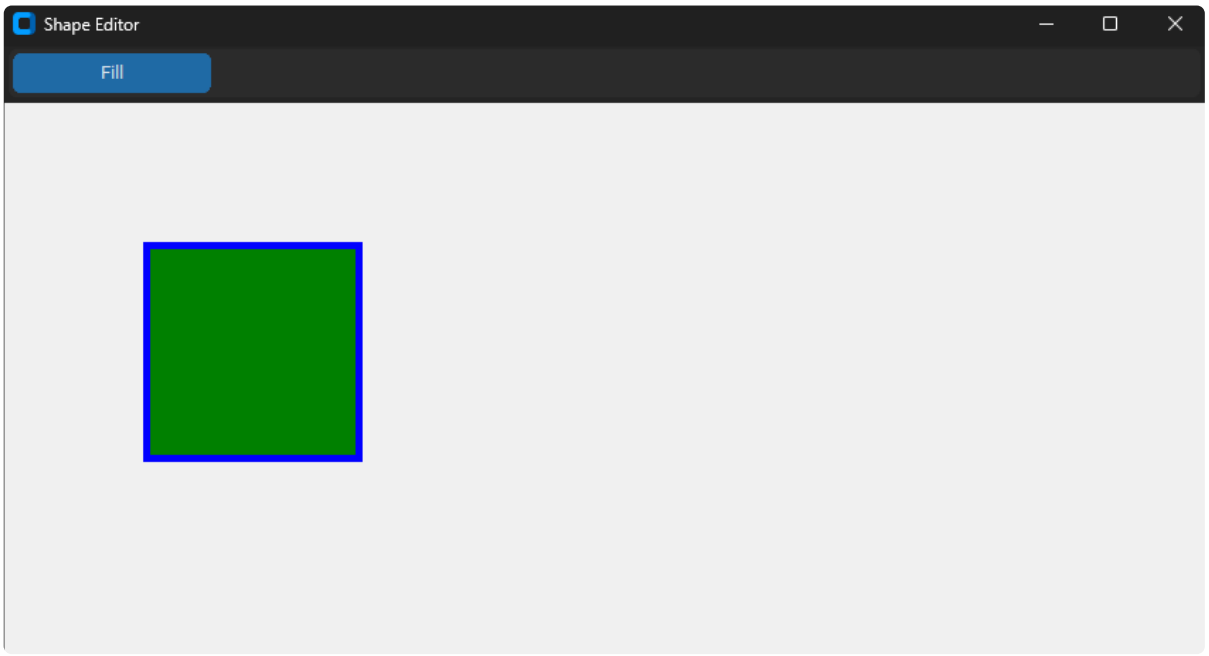
The signature for the button creation is:

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

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

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

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

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

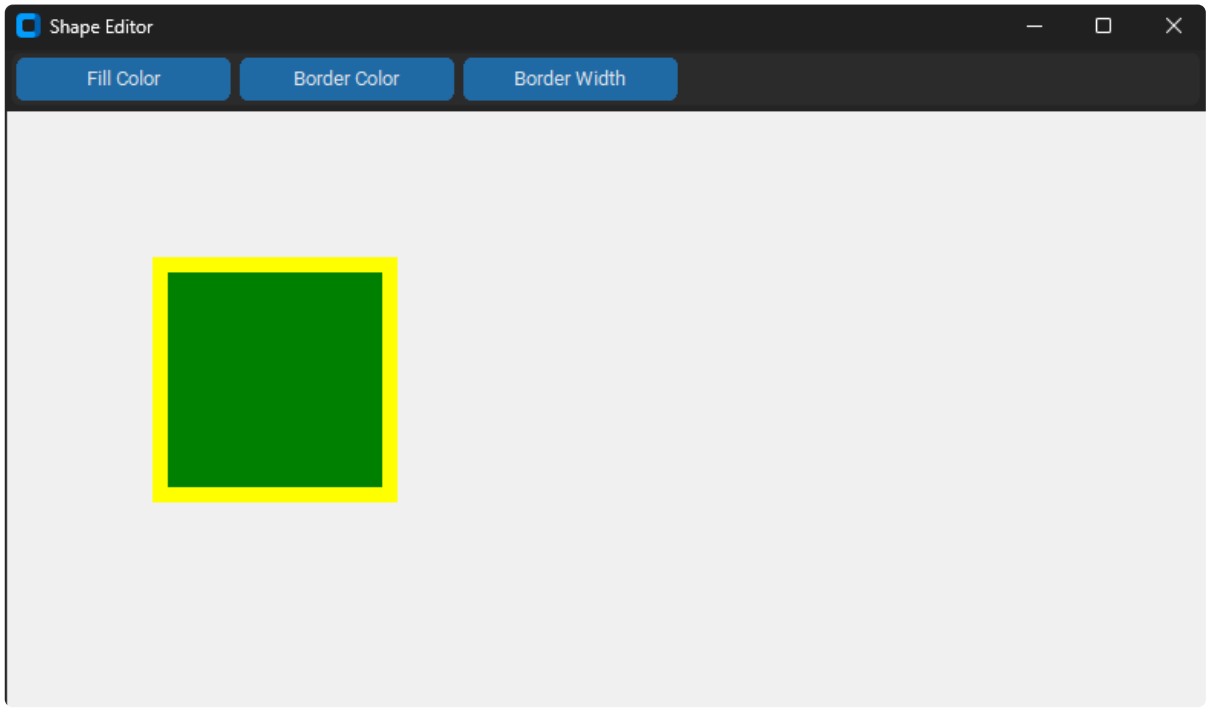
Shape Editor after the button is clicked. Note that the top frame has adjusted its height to the height of the button.



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

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

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

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

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

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

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

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



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

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

```
pip install CTkColorPicker
```

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

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

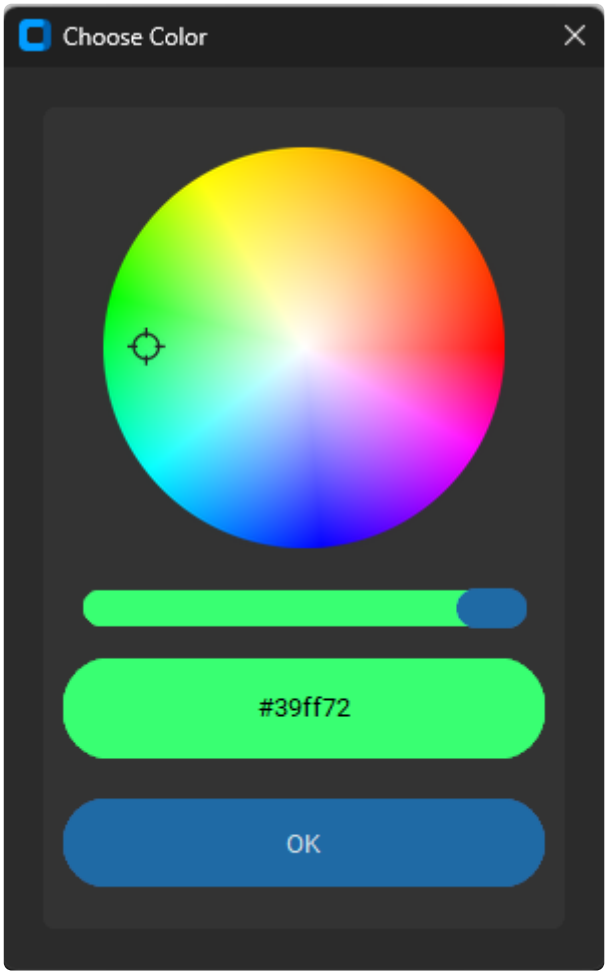
Import the color picker library.

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

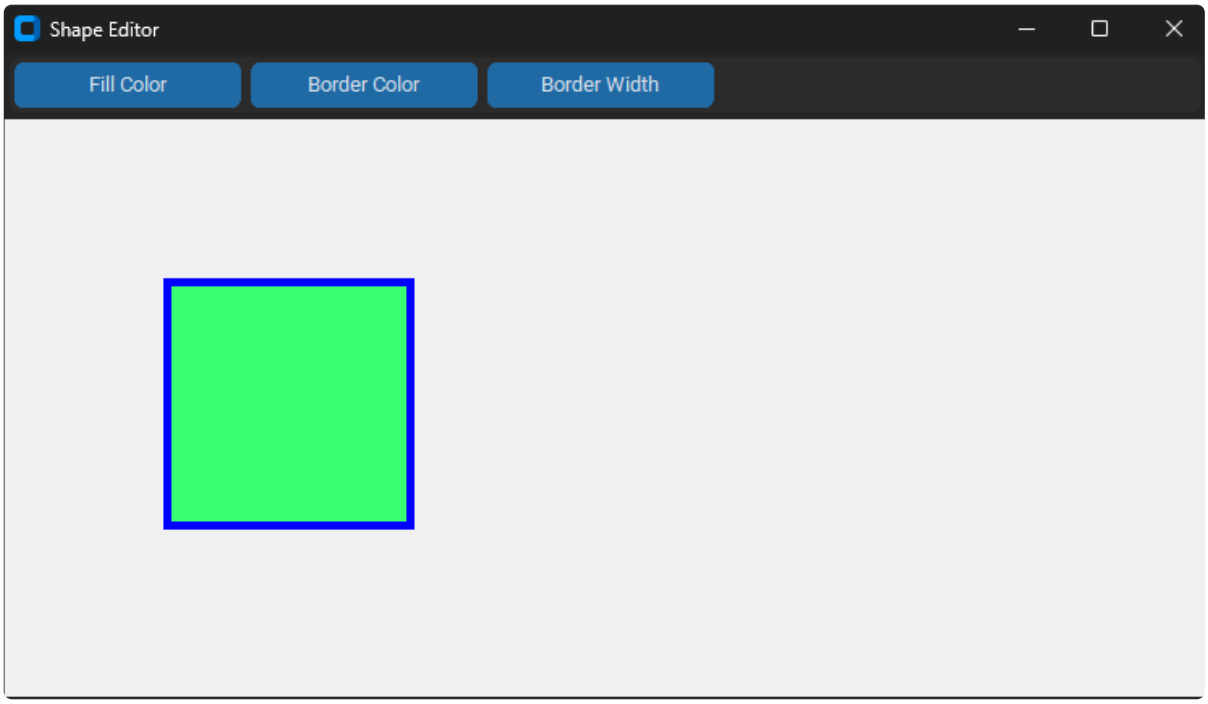
Add two lines of code to the color button functions that use the `AskColor()` function to open the color picker and `pick_color.get()` to store the user color. Then we set the fill color or the border color to the user selected color.

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

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

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



Rectangle after clicking the OK button in the color picker.



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

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

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

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

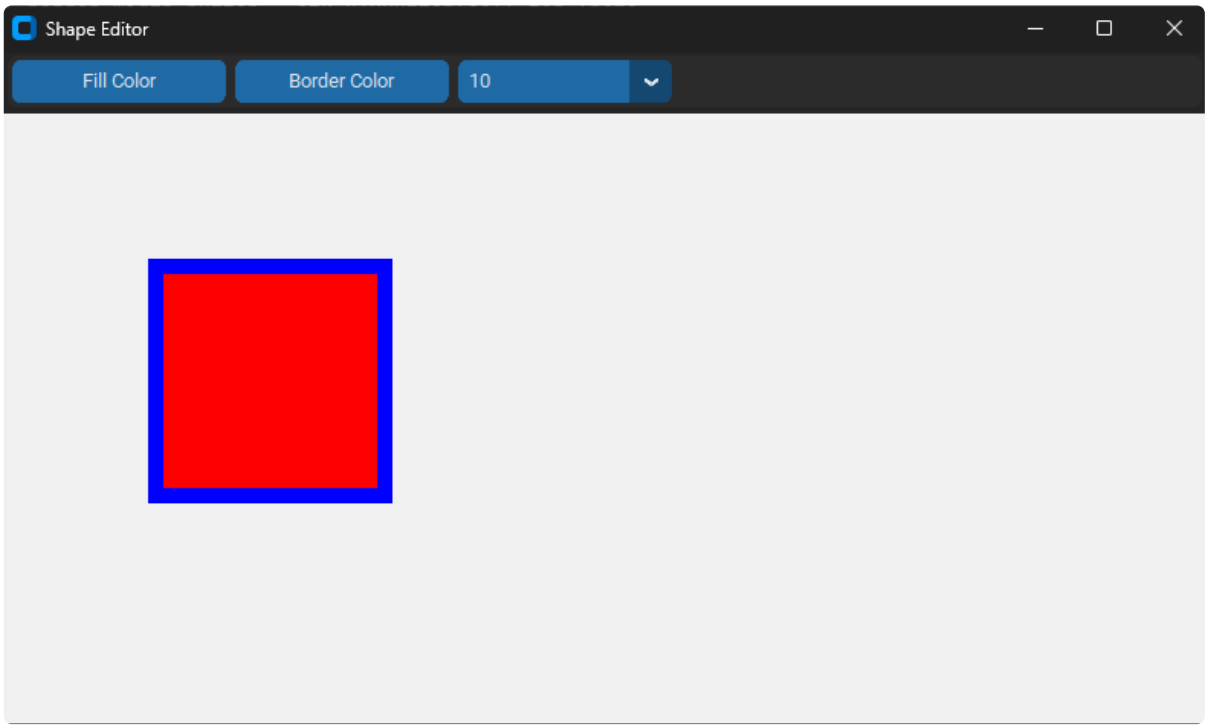
Note that I "commented out" the old code. This is a useful practice during refactoring until the new code is functioning properly as you might want to use some of the old code (like the pack options) or if the new code doesn't work you can delete it and uncomment the old code.

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

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

Main window with the border width option menu selected.

Rectangle with the border width set to 10 px.



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

Classes and Object-Oriented Programming (OOP)

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

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

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

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

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

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

Here is the complete class definition:

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

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

        self.id = None

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

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

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

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

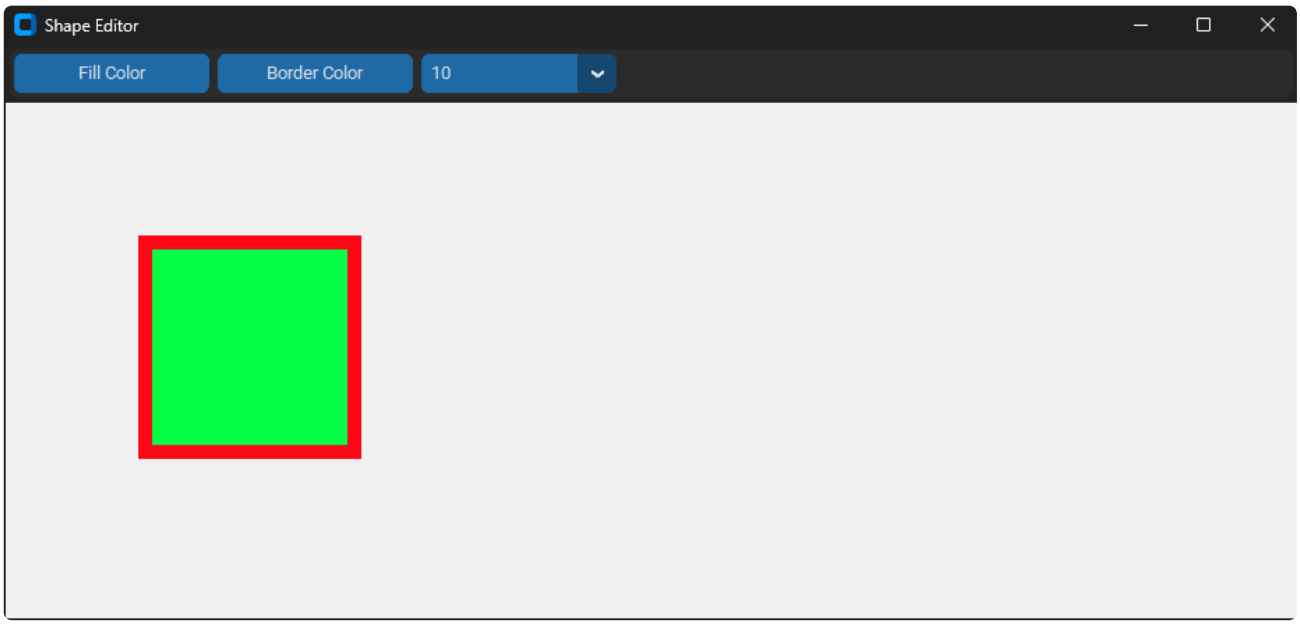
```
    def set_fill_color(self):
        pick_color = AskColor()  # open the color picker
        color = pick_color.get()  # get the color string
        self.canvas.itemconfig(self.rect_obj.id, fill=color)

    def set_border_color(self):
```

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

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

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

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

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

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

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



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

Oval class

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

        self.id = None

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

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

Updated Rectangle draw() method.

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

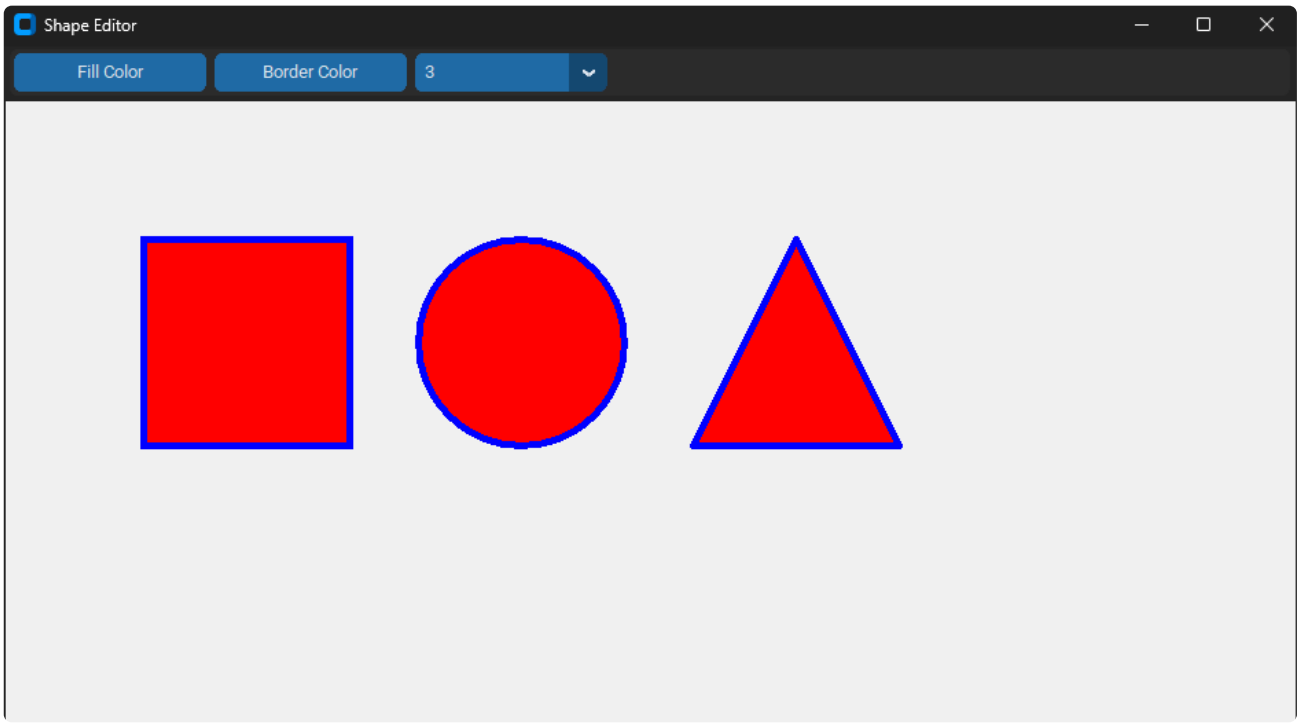
Triangle class

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

        self.id = None
        self.points = []

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

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

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

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

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

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

Run the program to see all three shapes



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

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

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

        self.id = None


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

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


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

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

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

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

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

Notice that the classes that inherit from the Shape class have the following statement; `super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color, border_width)` This passes the initialization arguments to the base class (Shape). Triangle keeps the `self.points` variable because it is unique to the Triangle class. This is a good example of how inheritance helps the programming eliminate duplicate code. Programmers call this the "DRY" principle, where DRY stands for Don't Repeat Yourself.

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

Shape Selection

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

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

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

        self.shape_list = []

        # Add widgets to the app here
```

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

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

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

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

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

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

What are the canvas mouse events and how do we get access? We need to create a canvas mouse binding to an event handler of the form:
`self.canvas.bind('<Button-1>', handler)`

The mouse buttons can be accessed with the following names:
Left mouse button: `"<Button-1>"`
Middle mouse button: `"<Button-2>"`
Right mouse button: `"<Button-3>"`

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

```
        border_width_optionmenu.set("3")

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

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

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

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

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

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

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

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

Mouse class

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

        self.selected_obj = None

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

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

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

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

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

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

```
        border_width_optionmenu.set("3")

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

    def set_fill_color(self):
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

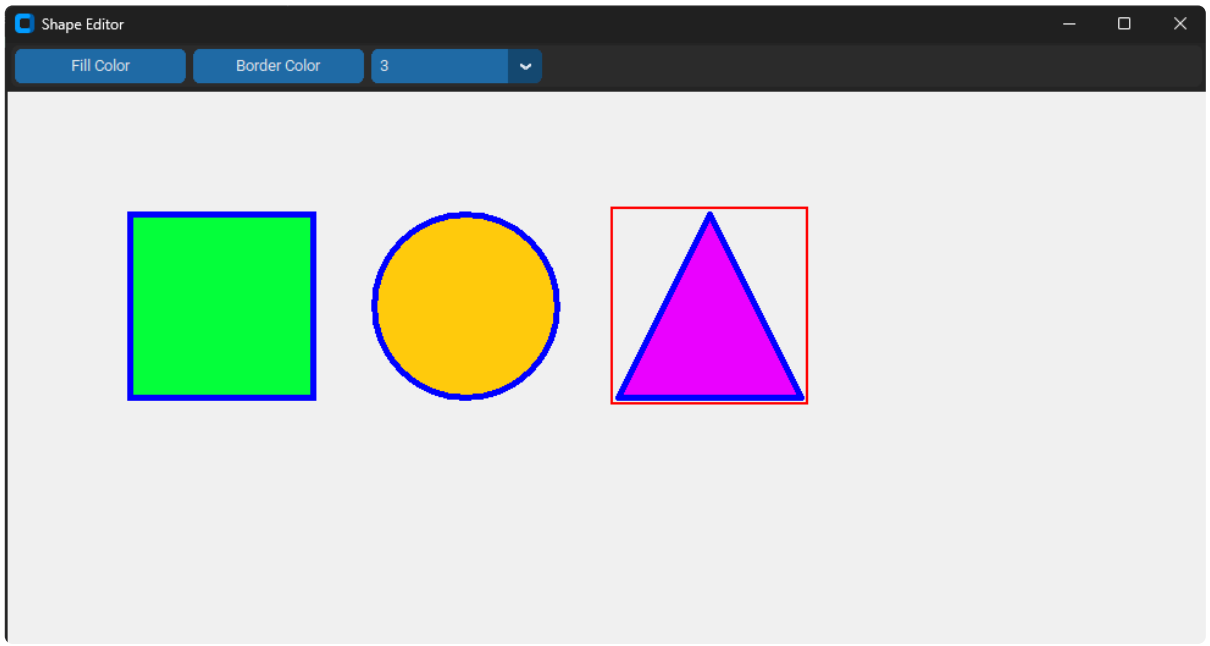
```
def draw(self):
    self.id = self.canvas.create_rectangle(self.x1, self.y1,
                                self.x2, self.y2,
                                fill=self.fill_color,
                                outline=self.border_color,
                                width=self.border_width)

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

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

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

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

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

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

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



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

Left Frame

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

Add a left frame to the Shape Editor App

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

        self.shape_list = []

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

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

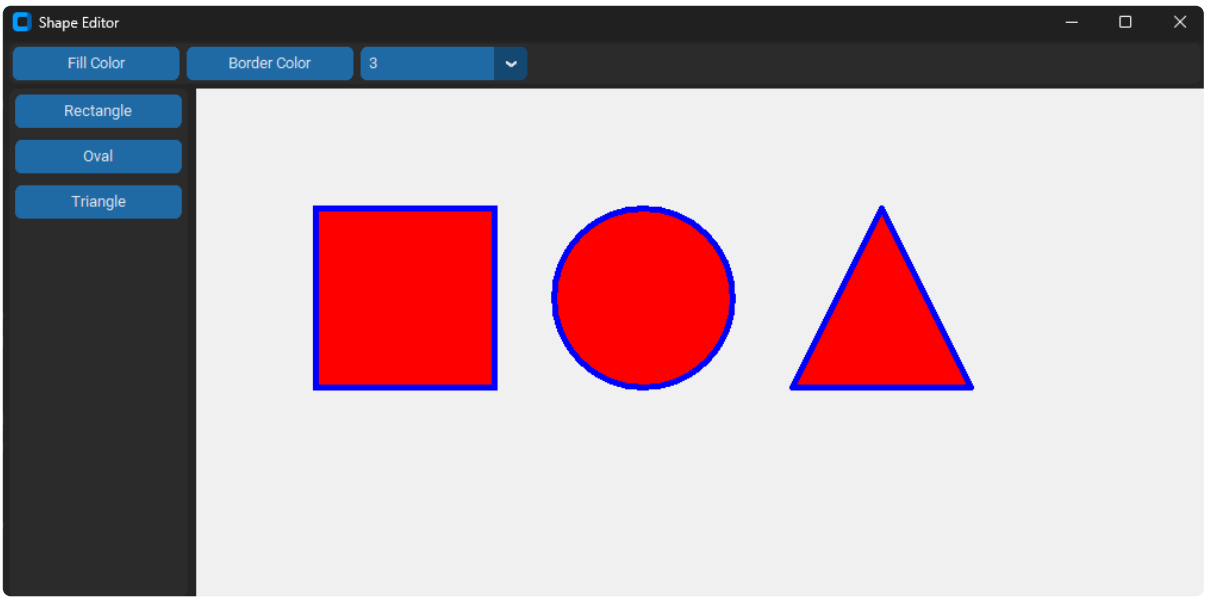
Next add three buttons to the frame for the Rectangle, Oval, and Triangle and add the button handles to create the shapes. Delete the old code that created the three shapes automatically.

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

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

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

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

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

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

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

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



Here is the current complete source code for shape_editory.py

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

import customtkinter as ctk
from CTkColorPicker import *


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

        self.id = None
        self.is_selected = False


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

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

        if self.is_selected:
```

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


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

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

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


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

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

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

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

```
width=2)


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

        self.selected_obj = None

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

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

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

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

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

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

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


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

        self.shape_list = []

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

Custom Canvas Class

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

Canvas class

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

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

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

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

Refactored Application Class

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

        self.shape_list = []

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Refactored Mouse Class

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

        self.selected_obj = None

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

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

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

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

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

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

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

Mouse Bindings

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

The project will implement four modes of operation:

- Draw new shapes with mouse

- Select and move
- Rotation
- Resize with mouse

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

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

        self.selected_obj = None

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

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

Draw New Shapes

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

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

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

def draw_bind_mouse_events(self):
```

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

    def draw_left_down(self, event):
        pass

    def draw_left_drag(self, event):
        pass

    def draw_left_up(self, event):
        pass
```

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

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

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

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

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

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

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

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

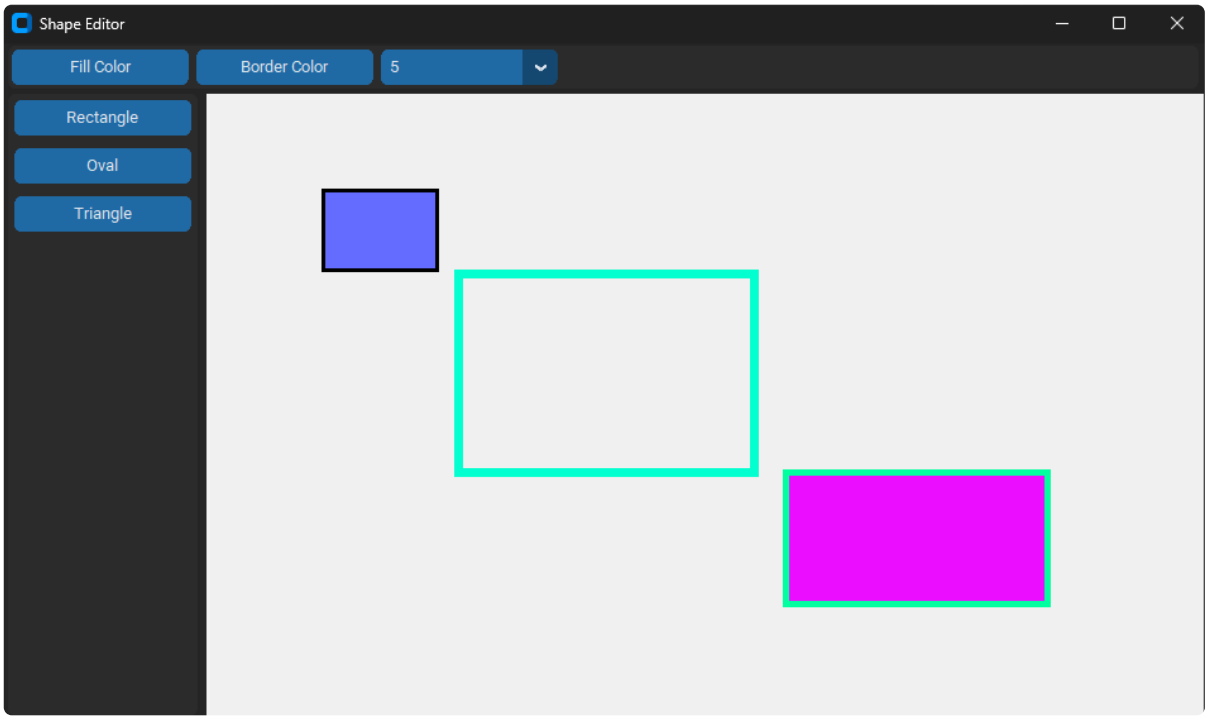
The draw left up method, resets the current shape and current shape obj variables, unbinds the draw mouse bindings and binds the left mouse down binding. This resets the bindings so that the new rectangle can be selected and the shape appearance controls work.

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

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

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

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



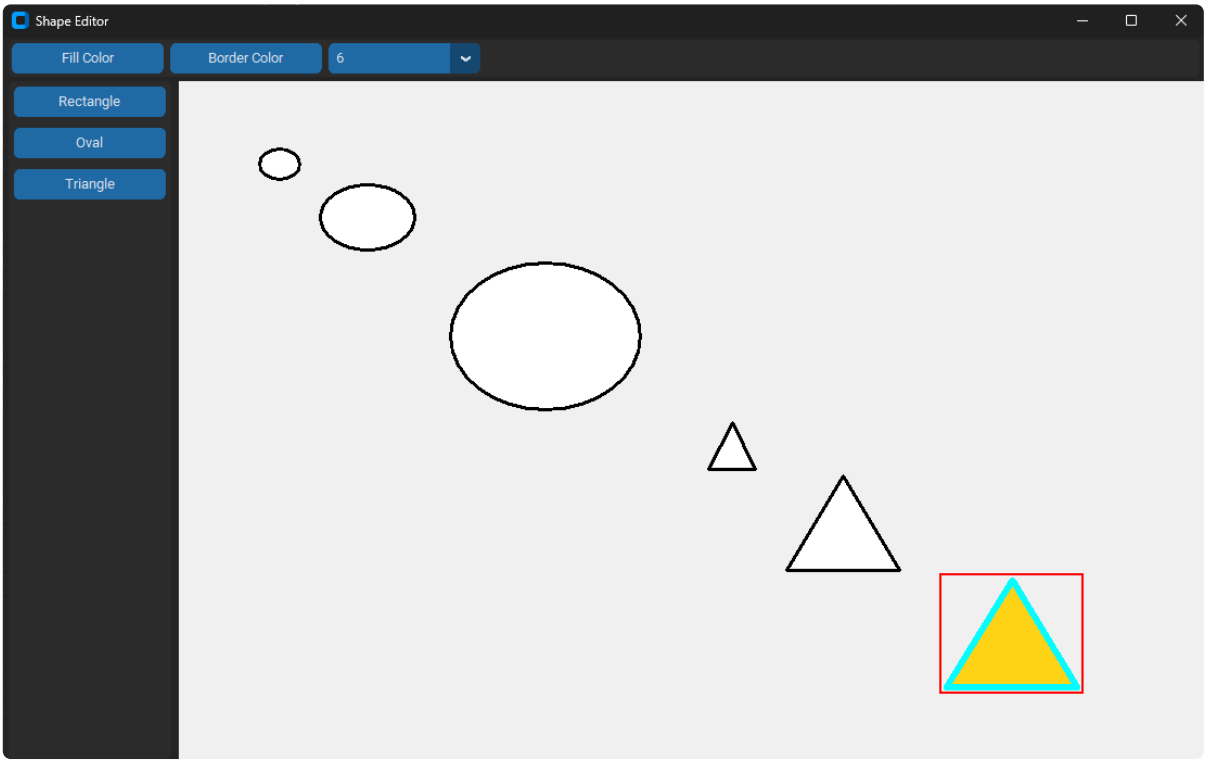
Implement the drag capability for ovals and triangles in the drag left down mouse event.

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

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

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

Update the shape menu button drivers for oval and triangle.

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

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



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

Move Shapes

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

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

    def move_left_down(self, event):
        pass

    def move_left_drag(self, event):
        pass

    def move_left_up(self, event):
        pass
```

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

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

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

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

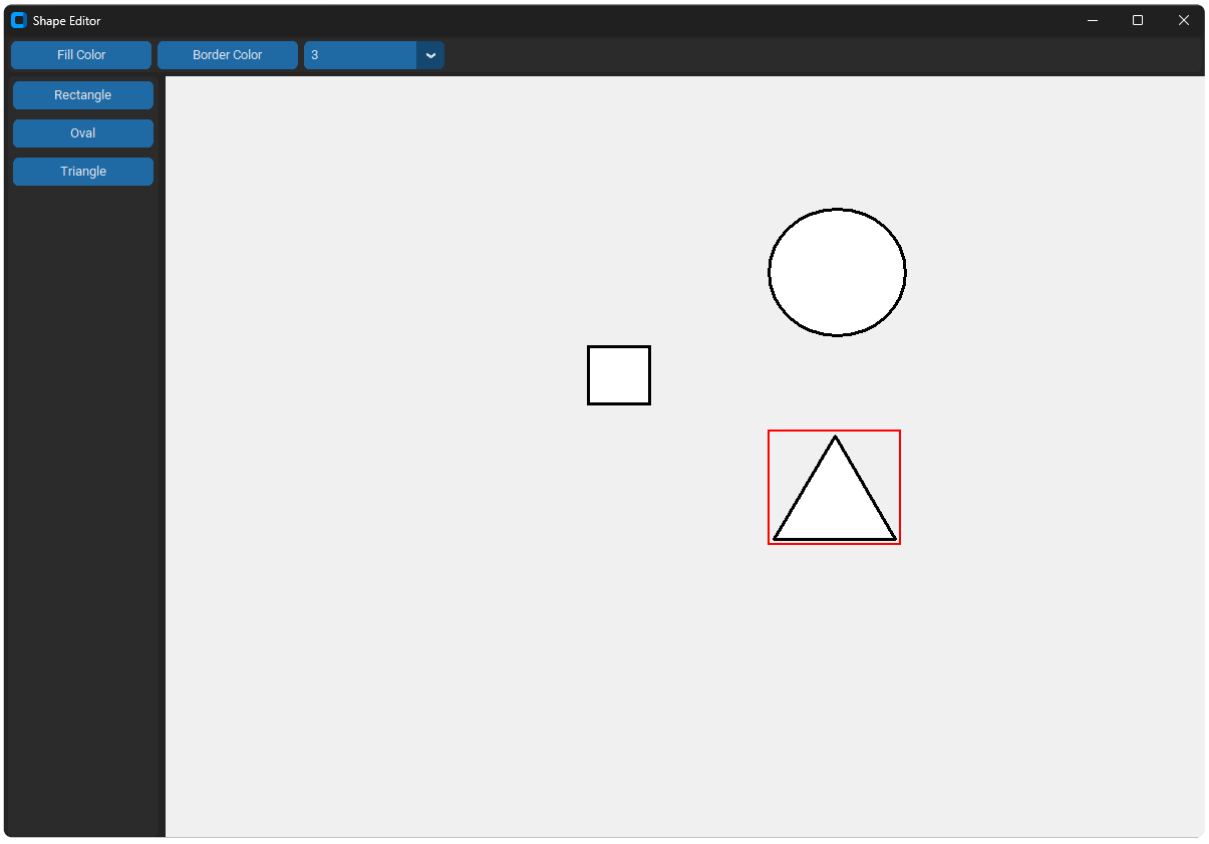
The move left drag method updates the shape coordinates with offset captured in the left mouse down method and redraws the canvas.

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

The move left up method resets the offset to 0.

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

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



Shape Rotation

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

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

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

        self.id = None
        self.is_selected = False
        self.angle = 0
```

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

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

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

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

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

        self.shape_list = []
```

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

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

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

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

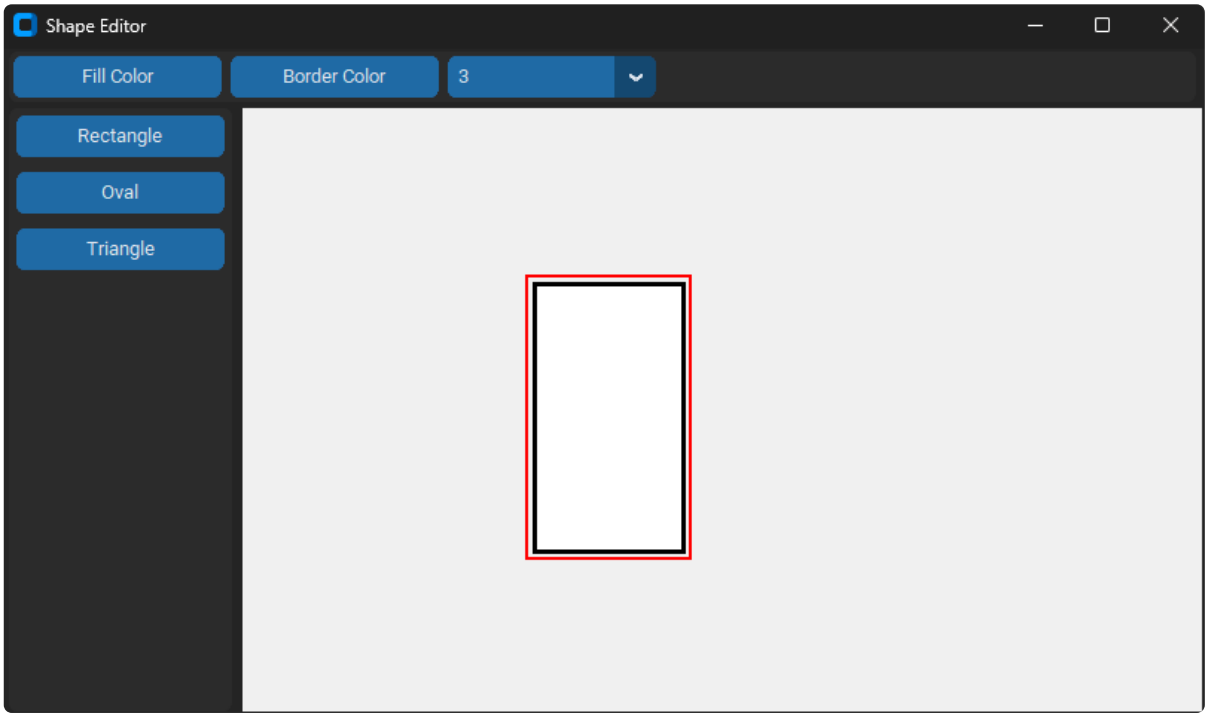
Rectangle Rotation

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

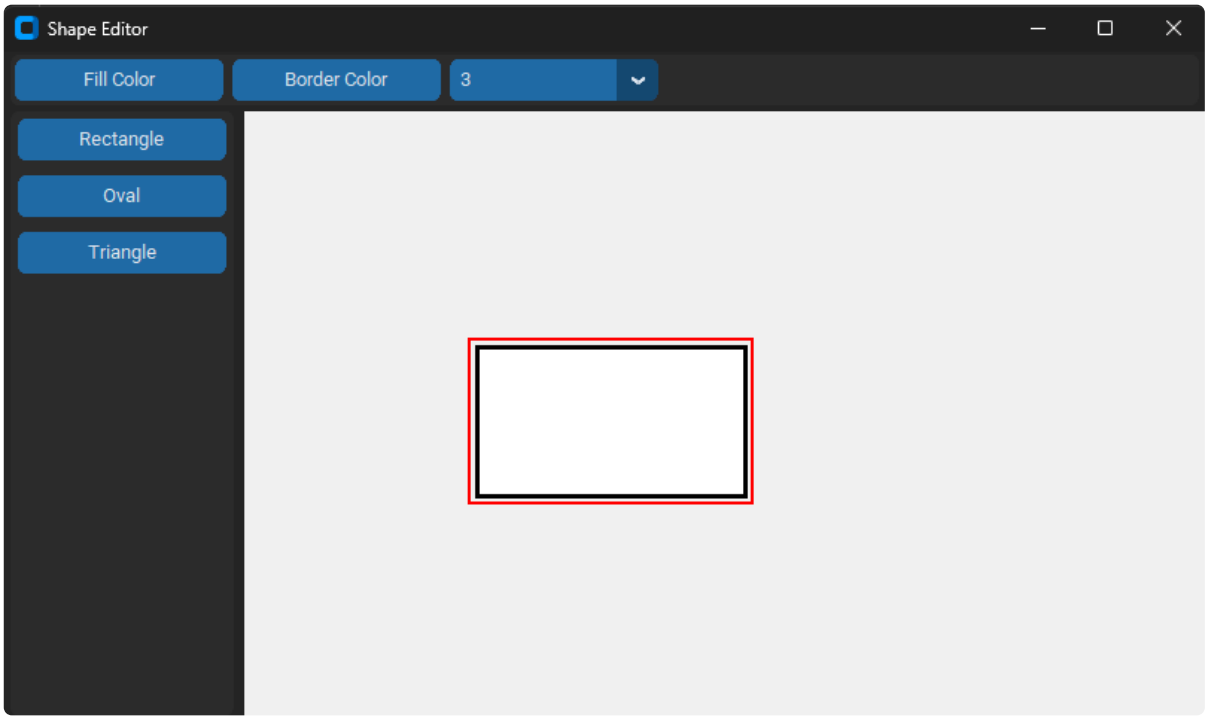
```
class Rectangle(Shape):
    def __init__(self, canvas, x1, y1, x2, y2, fill_color="white",
                 border_color="black", border_width=3):
        super().__init__(canvas, x1, y1, x2, y2, fill_color, border_color,
                 border_width)

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

```
                    center[0] + h/2, center[1] + w/2]
        else:
            points = [self.x1, self.y1, self.x2, self.y2]

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

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

Before hitting the 'r' key



After hitting the 'r' key



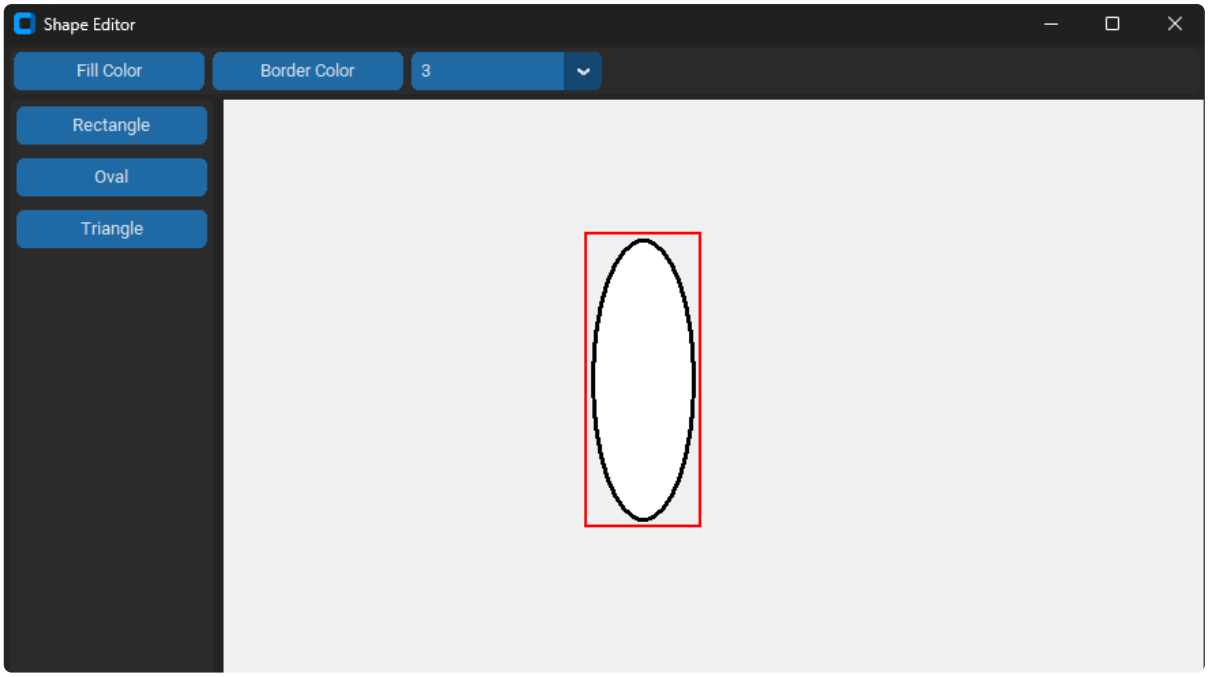
Oval Rotation

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

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

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

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

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

After rotation image

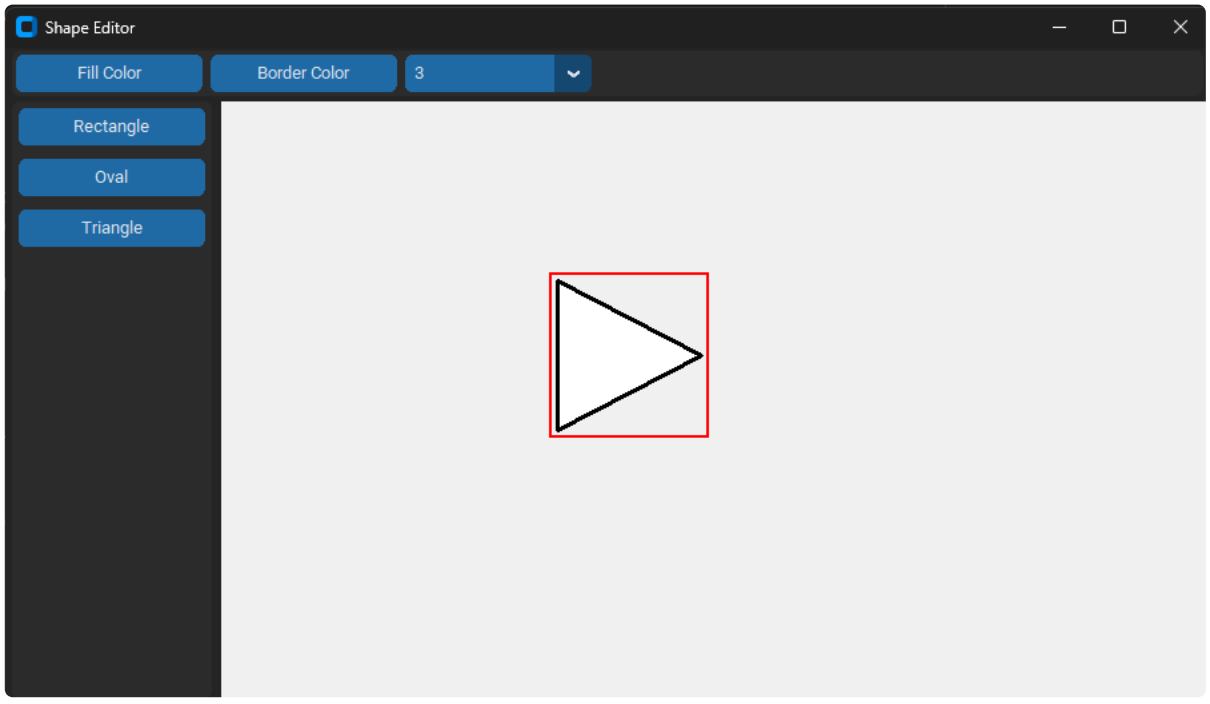


Triangle Rotation

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

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

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

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

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



Shape Resize

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

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

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

        self.id = None
        self.is_selected = False
        self.angle = 0
        self.scale = 1  # Default is 100% scale factor, range 0.1 to 10.0
```

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

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

Add the two new methods to the canvas class.

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

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

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

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

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

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

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

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

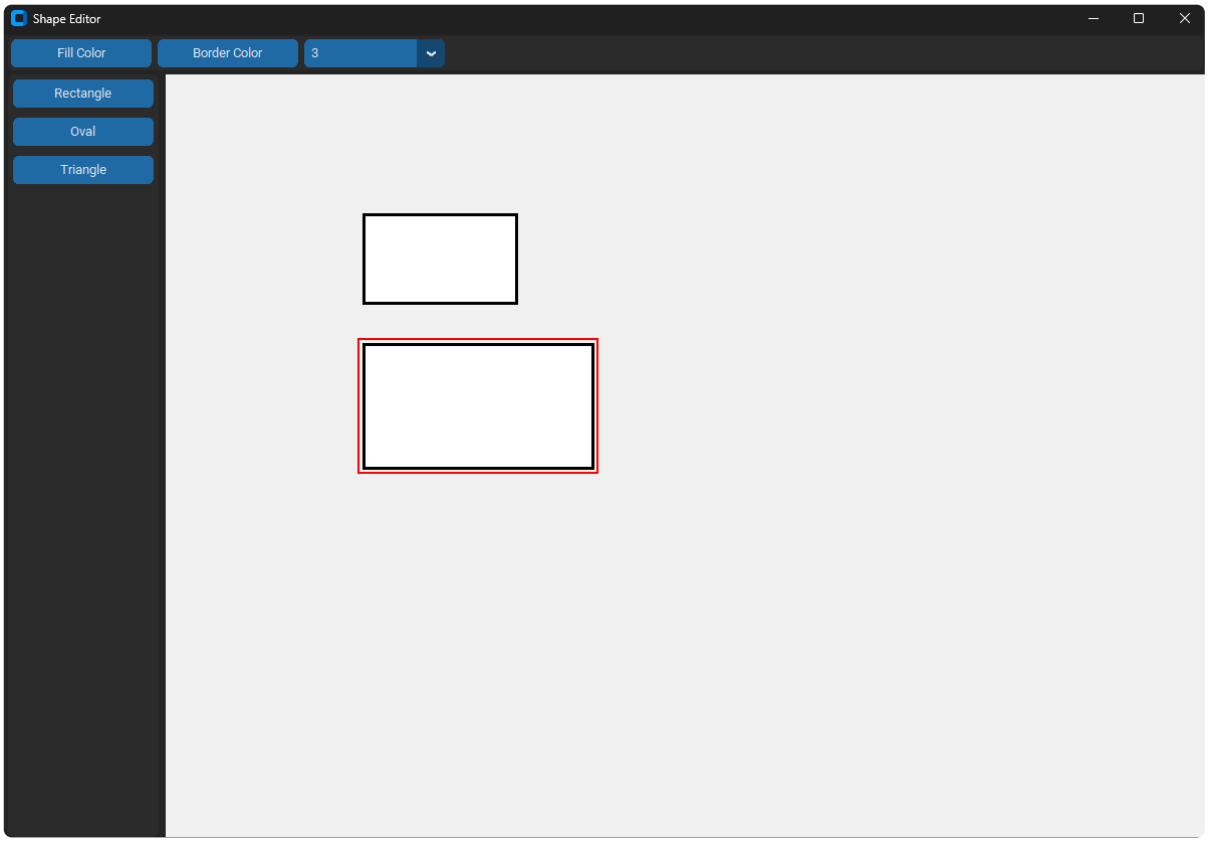
Rectangle Resize

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

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

    def draw(self):
```

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

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

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

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

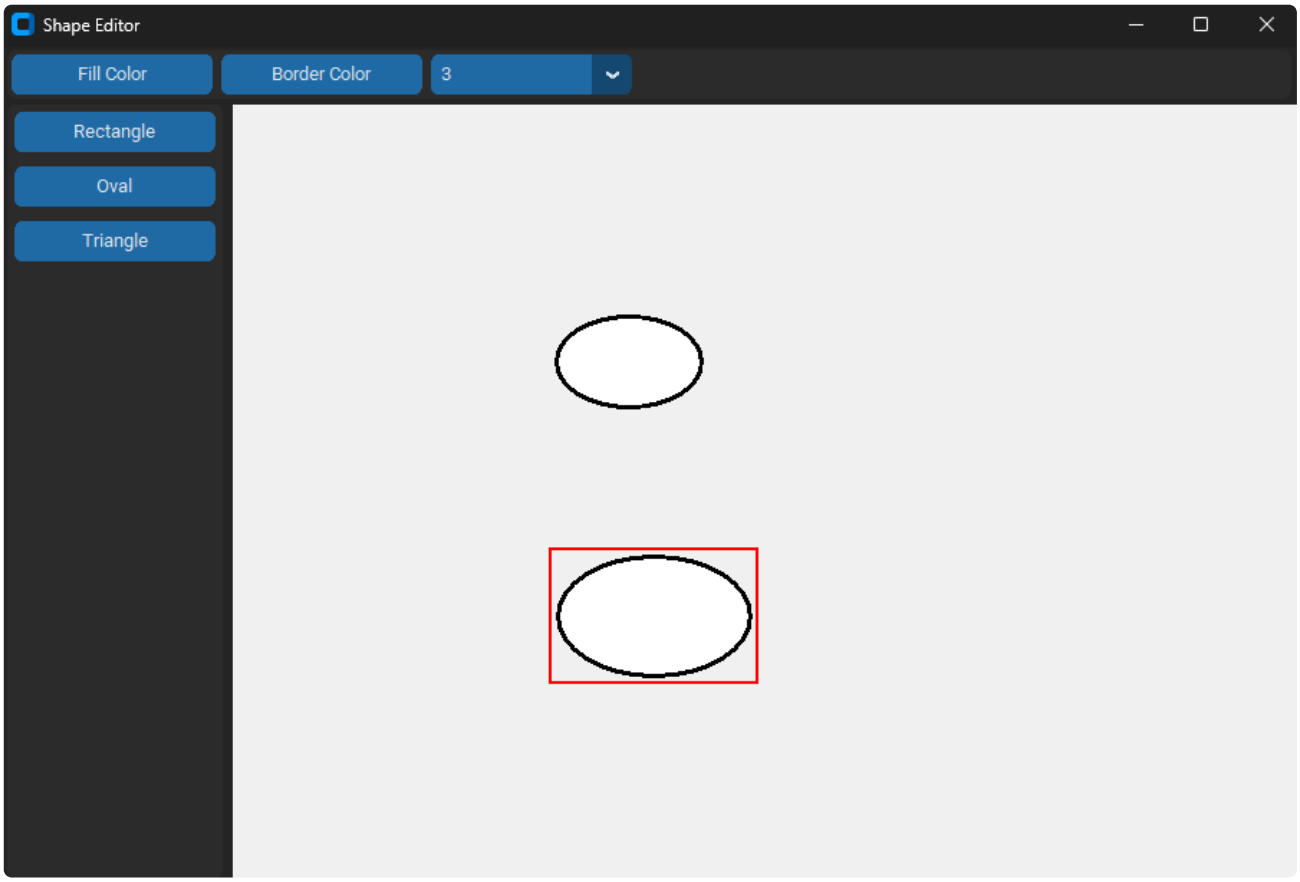


Oval Resize

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

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

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

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

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

Oval resize (scaling) test



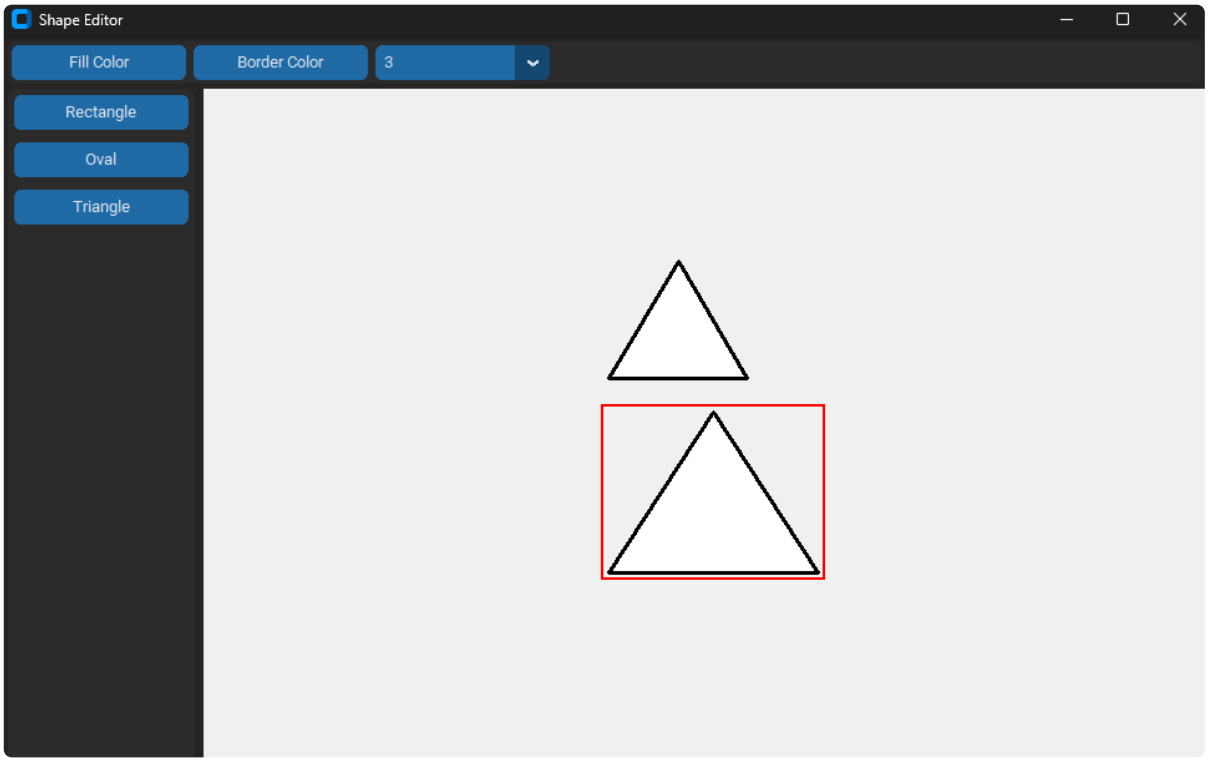
Triangle resize

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

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

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

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

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

Test Triangle resize (scaling)



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

Grid

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

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

Grid class

```
class Grid:
    type = "line"

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

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

            if Grid.type == "dot":
                self.dash_list = [1, 1]
            elif Grid.type == "line":
                self.dash_list = None

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

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

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

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

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

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

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

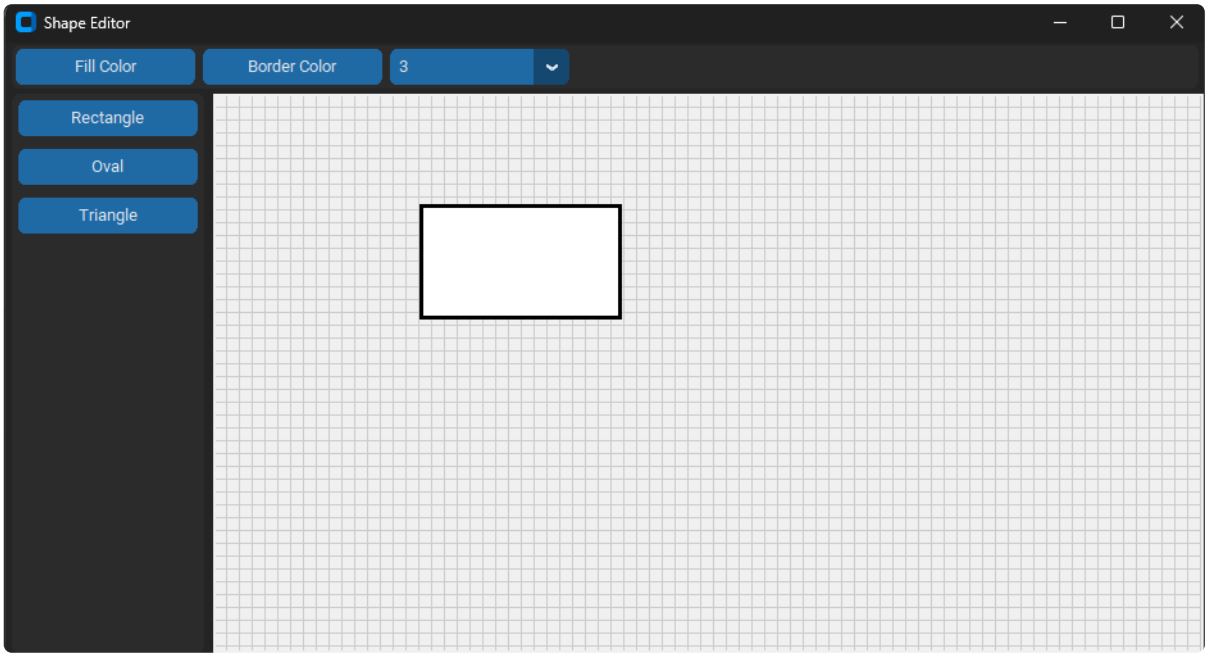
Update the canvas draw shapes method to draw the grid in the background. In the Canvas class initializer, set the grid size, create a Grid object call `self.grid`, and draw the grid.

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

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

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

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

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

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



Shape Snap-to-Grid

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

For example, in a mouse down event

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

where

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

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

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

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

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

Mouse draw event handlers

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

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

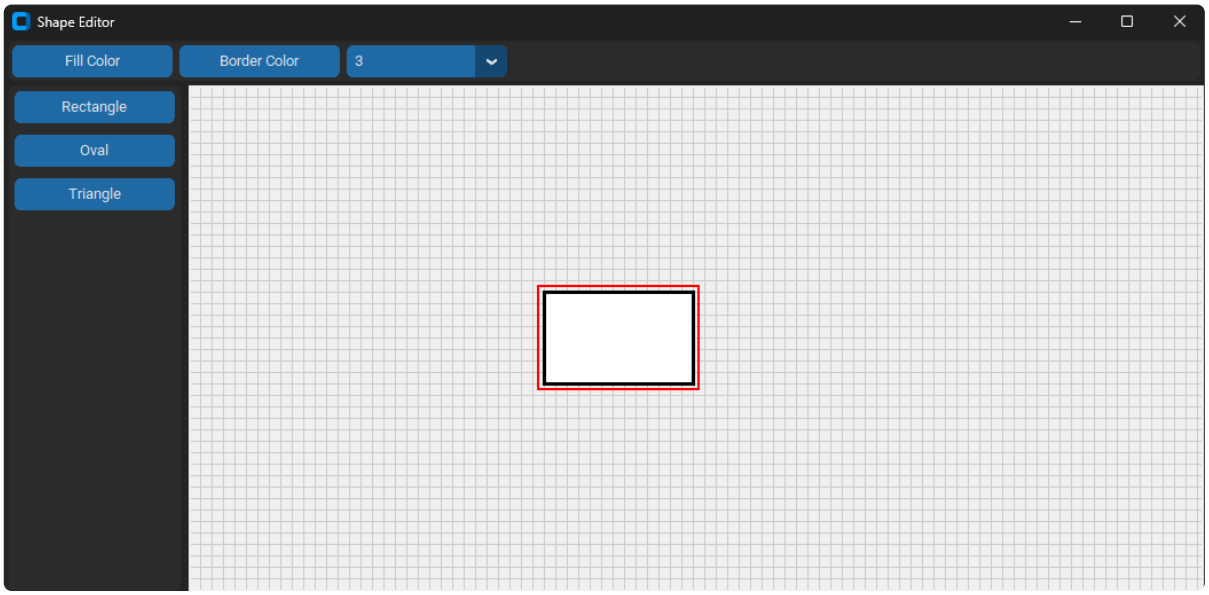
```
            self.start_x, self.start_y,
            fill_color="white",
            border_color="black",
            border_width=3)
    elif self.current_shape == "triangle":
        self.current_shape_obj = Triangle(self.canvas,
            self.start_x, self.start_y,
            self.start_x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

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

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

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

Mouse move event handlers

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

```
            self.canvas.draw_shapes()

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

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

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



Grid Visibility and Size

Add a ctk switch control to the top frame to control grid visibility. Add an option menu to the top frame to control grid size.

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

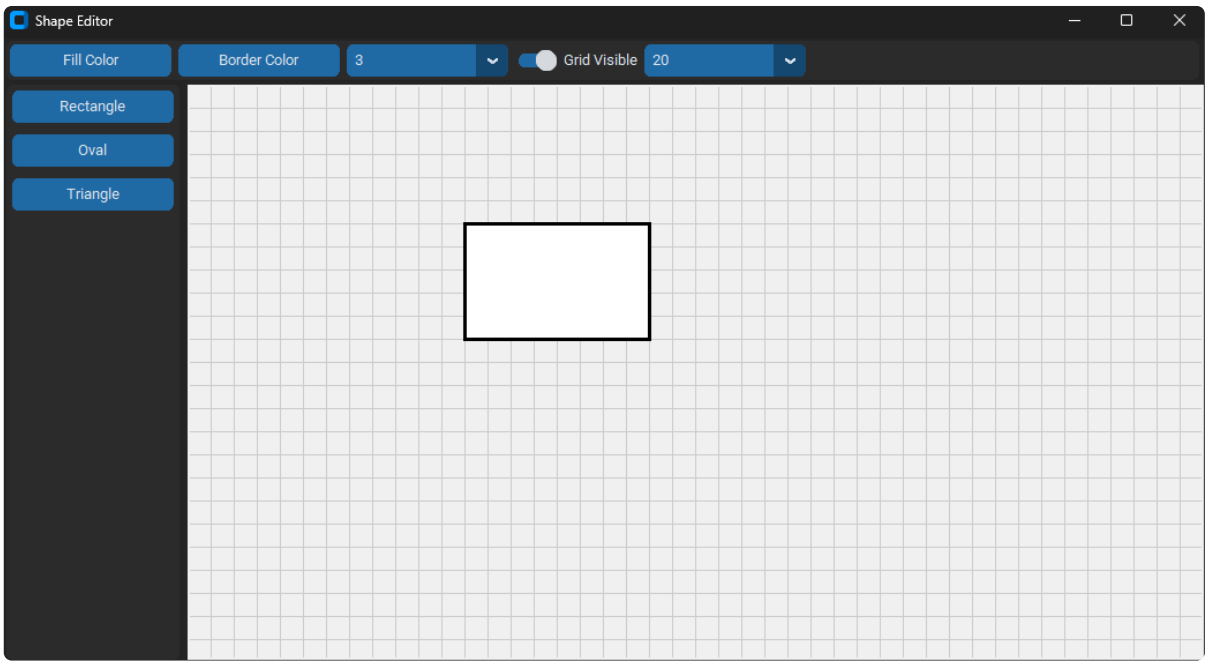
```
                                text="Fill Color",
                                command=self.set_fill_color)
        self.fill_color_button.pack(side=ctk.LEFT, padx=3, pady=3)

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

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

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

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

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

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

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


if __name__ == "__main__":
```

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

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



Modularization

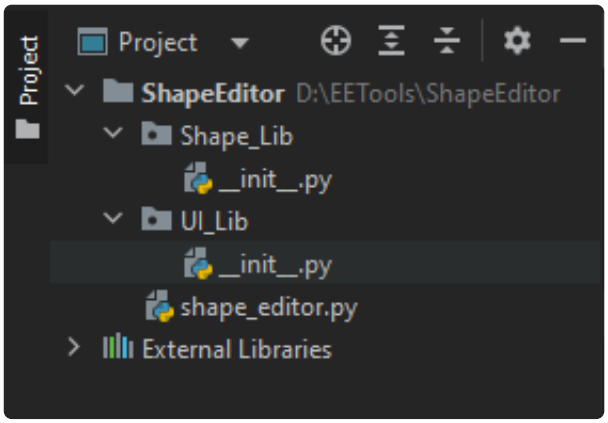
The application now has 460 lines of code. It is time to refactor it to creates smaller, manageable modules and packages.

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

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

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

Directory structure



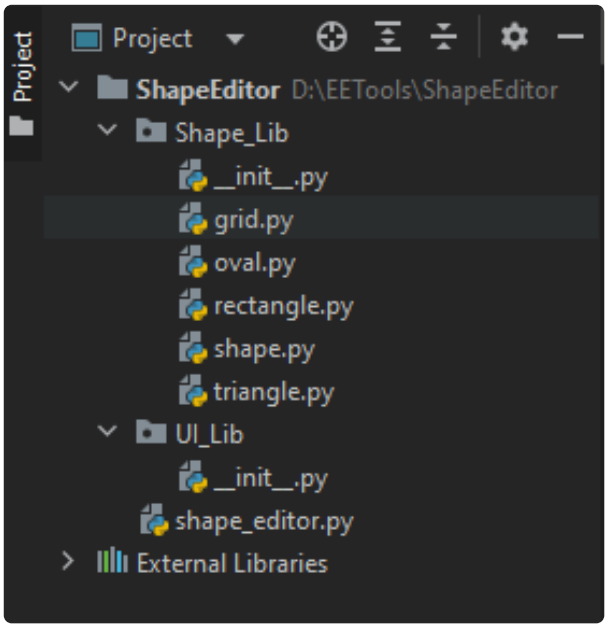
Start with the shape classes and create a separate file for each class:

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

We need to tell the Rectangle, Oval, and Triangle classes where to find the Shape class. Add the following line to the top of each sub-class:

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

Directory structure after modularization of the shape classes



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

```
import customtkinter as ctk
from CTkColorPicker import *

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

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

Next move UI classes to the UI_Lib directory

- Canvas class
- Mouse class

Mouse Module

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

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


class Mouse:
```

Canvas Module

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

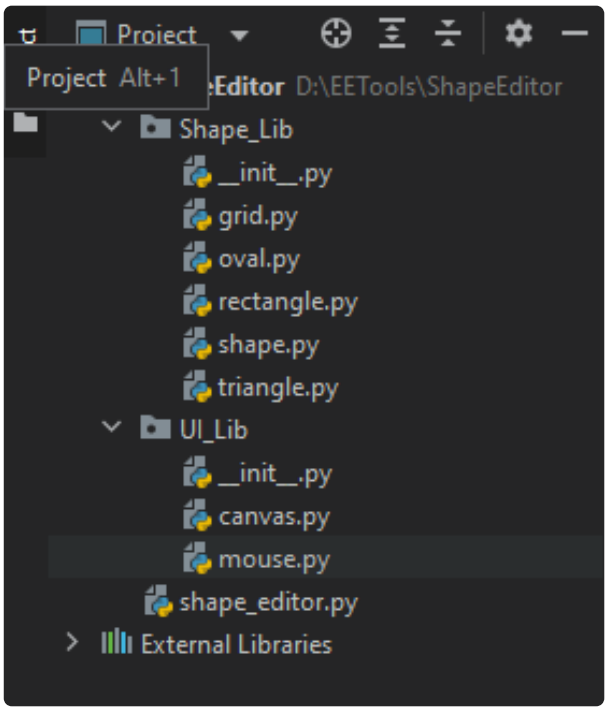
```
import customtkinter as ctk
from UI_Lib.mouse import Mouse
from Shape_Lib.grid import Grid


class Canvas(ctk.CTkCanvas):
```

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

```
import customtkinter as ctk
from CTkColorPicker import *

from UI_Lib.canvas import Canvas


class ShapeEditorApp(ctk.CTk):
```

Run the program and verify that it runs without error.

Directory structure



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

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

Top Frame Module

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

```
import customtkinter as ctk
from CTkColorPicker import *


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

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

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

```
import customtkinter as ctk
from CTkColorPicker import *


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

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

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

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

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

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

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

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

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

    def grid_visibility(self):
```

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

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

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

```
import customtkinter as ctk

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


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

        self.shape_list = []

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

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

Left Frame Module

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

```
import customtkinter as ctk


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

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

```
import customtkinter as ctk


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

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

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

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

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

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

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

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

```
import customtkinter as ctk

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


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

        self.shape_list = []

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

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

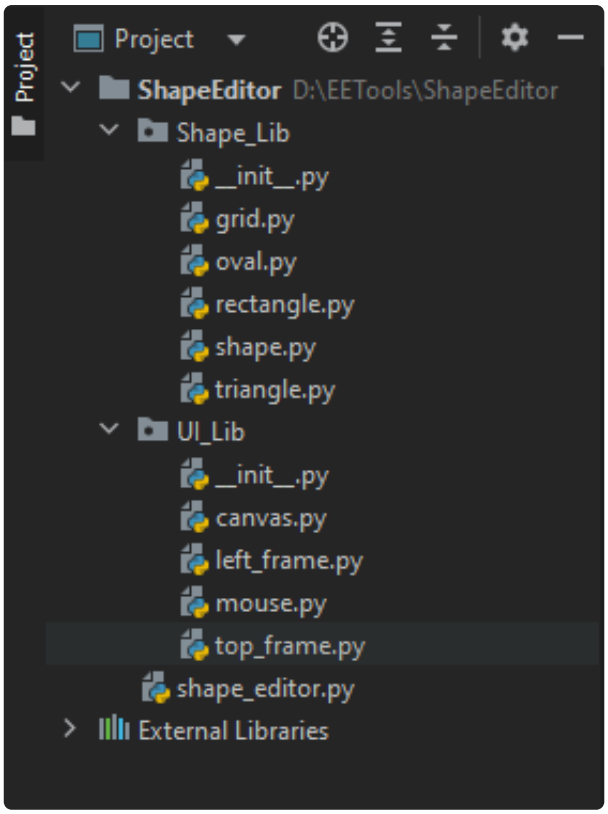
Summary

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

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

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

Final directory structure



shape_editor.py

```
import customtkinter as ctk

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


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

```
        self.shape_list = []

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

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

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

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


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

UI_Lib directory:

Empty `__init__.py` file

canvas.py

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


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

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

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

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

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

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

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

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

left_frame.py

```
import customtkinter as ctk
```

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

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

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

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

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

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

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

mouse.py

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


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

```
        self.canvas = canvas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

top_frame.py

```
import customtkinter as ctk
from CTkColorPicker import *


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

        # Add top frame widgets here
        self.fill_color_button = ctk.CTkButton(self,
                                    text="Fill Color",
```

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

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

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

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

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

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

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

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

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

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

Shape_Lib directory:

Empty `__init__.py` file

grid.py

```
class Grid:
    type = "line"

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

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

            if Grid.type == "dot":
                self.dash_list = [1, 1]
            elif Grid.type == "line":
                self.dash_list = None

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

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

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

oval.py

```
from Shape_Lib.shape import Shape


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

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

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

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

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

rectangle.py

```
from Shape_Lib.shape import Shape


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

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

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

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

shape.py

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

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

triangle.py

```
from Shape_Lib.shape import Shape


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

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

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

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

On to the Line Editor Project.