Project Deliverable #3

Potential Features

- Add Checkboxes to Legend Controlling Subplot Visibility #16796
  - https://github.com/matplotlib/matplotlib/issues/16796
  - See "*i16796.pdf*" for further explanation
- Add scrollbars when needed to preserve specified figure size #7338
  - https://github.com/matplotlib/matplotlib/issues/7338
  - See "*i7338.pdf*" for further explanation

Selected Feature: #16796

The Two Features

The two features we decided to work on is the Legend feature (#16796) and the Scroll bar (#7338). The legend feature is adding a checkbox in the legend allowing the user to keep certain sub plots on the graph hidden or not. The scroll bar feature allows the user to be able to move the window screen vertically or horizontally if the canvas is bigger than the main window screen so they can see the whole figure.

Challenge Factors for Each Feature

- #16796
  - Figuring out how to add the checkbox image to the GUI
  - Making the checkbox image swap from with or without checkmark upon being clicked
  - Learning the matplotlib artist abstract class and creating a new GUI object
  - Learning mouse events; how to add and remove them
- #7338
  - Adding and removing the scroll bar from the MainWindow

Feature to Implement

From reviewing challenges of each feature, we decided to choose the Legend feature (#16796). First, it was clear from above that the Legend feature would present more challenges. The scroll bar GUI was already implemented (come packaged with PyQt) so that would be a big part we wouldn't get to do if we chose that feature. Another reason is that the widgets are in a separate package than matplotlib, meaning we'd have to also learn how to interact with their objects. This wouldn't be difficult, but it would require learning something that we'd have nothing to show for. Lastly, the UML diagram for the legend feature will look better, adding our own class allows us to show what new interactions there will be, where as the scrollbar is very contained and won't have the same quality of interactions.

Acceptance Testing

Test 1: checkbox=False

When the legend is initialized, if the argument checkbox is given the keyword False then it shouldn't produce an interactive legend.

Steps to Reproduce

1. Set up the matplotlib repo with our new feature, instructions will be in deliverable 4's README.
2. Create and run a python file that creates a figure, adds elements with labels, and creates a legend where the argument checkbox is given the value False.
    - an example file, *test1.py*, is provided in the Acceptance_Testing directory
3. Verify the following
    - In the figure all elements that were created are visible
    - In the figure a non-interactive legend is visible

Test 2: checkbox=True

When the legend is initialized, if the argument checkbox is given the keyword True then it will produce an interactive legend.

Steps to Reproduce

1. Set up the matplotlib repo with our new feature, instructions will be in deliverable 4's README
2. Create and run a python file that creates a figure, adds elements with labels, and creates a legend where the argument checkbox is given the value True.
   - an example file, *test2.py*, is provided in the Acceptance_Testing directory
3. Verify the following
   - In the figure all elements that were created are visible
   - In the figure an interactive legend is visible
      i. Each entry has a corresponding checkbox
      ii. Each checkbox has a checkmark by default
      iii. Clicking on a checkbox with a checkmark will remove the checkmark and will remove the corresponding element from the figure
      iv. Clicking on an empty checkbox will add a checkmark to it and will add the corresponding element back to the figure

Project Architecture

While working on fixing issues and discussing new features, we noticed quite a few design patterns in the architecture; the more interesting ones are detailed below.

Strategy Design Pattern

In the backend section of the architecture, a variety of interchangeable backends are defined and encapsulated. The user can use "use" function to switch between the backends easily. All these backends implements an abstract class RendererBase which defines some abstract methods that must be implemented to ensure full functionality including draw_path, draw_image, and draw_gouraud_triangle.

Decorator Design Pattern

In legend.py, DraggableLegend is a wrapper around a `.Legend` to support mouse dragging. We recognize this to be a decorator design pattern because it lets you attach new behaviors to objects by wrap the objects in wrapper objects that contain the behaviors.

Singleton Design Pattern

In _pylab_helpers.py, the Gcf class is a singleton to maintain the relation between figures and their managers, and to keep track of the "active" figure and its manager. We recognize this class to be singleton design pattern because there is just one instance of current figure.

Iterator Design Pattern

In __init__.py in the cbook module, the Grouper class utilizes iterator design pattern. This class provides a lightweight way to group arbitrary objects together into disjoint sets when a full-blown graph data structure would be overkill. The Grouper object can be used as an iterator. It lets you traverse objects of without exposing its underlying representation.

For an overview of how the Legend feature will interact with the overall system, please view “i16796_uml.html”.