Project Report: Manim

Name: Nick
System Under Test (SUT): manim
Link to SUT Source Code: https://github.com/3b1b/manim
Link to all of my contributions:
https://github.com/csun-comp587-s20/manim/tree/testing/test
Capabilities Under Test:

- **MObject** is **Precise:** MObjects hold the exact mathematical function they describe, as well as instructions to visualize said function
- **MObject** is **Consistent:** the visualization instructions held by an MObject yield a single, unique animation

Unit Testing with Sufficient Coverage:
For unit testing, I proposed to test the Mobject file to assert that it behaved in a precise and consistent manner. This involved asserting that Mobjects can add/remove children to create complex objects. Additionally, the majority of tests center around transforming the points a Mobject and its submobjects hold.

Link to unit tests:
https://github.com/csun-comp587-s20/manim/blob/testing/test/test_mobject.py

As for line coverage, the file currently sits at 51% coverage, and the SUT as whole is at 32%. While there are clearly more tests that need to be written, most of these tests would be for drawing calls, coloring functions, and setters/getters. Of the 1100 lines in the Mobject file, the first 600 lines are the most relevant to the capabilities being tested.

Evidence:

Coverage for **manimlib/mobject/mobject.py** : 52%

706 statements | 365 run | 341 missing | 0 excluded

Coverage report: 32%

Module	*statements* ↑	*missing*	*excluded*	*coverage*
manimlib/mobject/mobject.py	706	341	0	52%
manimlib/mobject/types/vectorized_mobject.py	537	419	0	22%
manimlib/mobject/geometry.py	443	293	0	34%

```
    # Transforming operations

    def apply_to_family(self, func):
        for mob in self.family_members_with_points():
            func(mob)

    def shift(self, *vectors):
        total_vector = reduce(op.add, vectors)
        for mob in self.family_members_with_points():
            mob.points = mob.points.astype('float')
            mob.points += total_vector
        return self

    def scale(self, scale_factor, **kwargs):
        """
        Default behavior is to scale about the center of the mobject.
        The argument about_edge can be a vector, indicating which side of
        the mobject to scale about, e.g., mob.scale(about_edge = RIGHT)
        scales about mob.get_right().

        Otherwise, if about_point is given a value, scaling is done with
        respect to that point.
        """
        self.apply_points_function_about_point(
            lambda points: scale_factor * points, **kwargs
        )
        return self

    def rotate_about_origin(self, angle, axis=OUT, axes=[]):
        return self.rotate(angle, axis, about_point=ORIGIN)

    def rotate(self, angle, axis=OUT, **kwargs):
        rot_matrix = rotation_matrix(angle, axis)
        self.apply_points_function_about_point(
            lambda points: np.dot(points, rot_matrix.T),
            **kwargs
        )
        return self

    def flip(self, axis=UP, **kwargs):
        return self.rotate(TAU / 2, axis, **kwargs)

    def stretch(self, factor, dim, **kwargs):
        def func(points):
            points[:, dim] *= factor
            return points
        self.apply_points_function_about_point(func, **kwargs)
        return self
```

```
    # Positioning methods

    def center(self):
        self.shift(-self.get_center())
        return self

    def align_on_border(self, direction, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
        """
        Direction just needs to be a vector pointing towards side or
        corner in the 2d plane.
        """
        target_point = np.sign(direction) * (FRAME_X_RADIUS, FRAME_Y_RADIUS, 0)
        point_to_align = self.get_critical_point(direction)
        shift_val = target_point - point_to_align - buff * np.array(direction)
        shift_val = shift_val * abs(np.sign(direction))
        self.shift(shift_val)
        return self

    def to_corner(self, corner=LEFT + DOWN, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
        return self.align_on_border(corner, buff)

    def to_edge(self, edge=LEFT, buff=DEFAULT_MOBJECT_TO_EDGE_BUFFER):
        return self.align_on_border(edge, buff)
```

```
# Color functions

def set_color(self, color=YELLOW_C, family=True):
    """
    Condition is function which takes in one arguments, (x, y, z).
    Here it just recurses to submobjects, but in subclasses this
    should be further implemented based on the the inner workings
    of color
    """
    if family:
        for submob in self.submobjects:
            submob.set_color(color, family=family)
    self.color = color
    return self

def set_color_by_gradient(self, *colors):
    self.set_submobject_colors_by_gradient(*colors)
    return self

def set_colors_by_radial_gradient(self, center=None, radius=1, inner_color=WHITE, outer_color=BLACK):
    self.set_submobject_colors_by_radial_gradient(
        center, radius, inner_color, outer_color)
    return self

def set_submobject_colors_by_gradient(self, *colors):
    if len(colors) == 0:
        raise Exception("Need at least one color")
    elif len(colors) == 1:
        return self.set_color(*colors)

    mobs = self.family_members_with_points()
    new_colors = color_gradient(colors, len(mobs))

    for mob, color in zip(mobs, new_colors):
        mob.set_color(color, family=False)
    return self
```

Automated Testing:

For automated testing, I chose to implement a property based approach because of the very 'mathy' nature of the library. To do so, I implemented a Mobject generator that would generate Mobjects up to a maximum depth, with a maximum number of children, a maximum number of points, and with all of the points on the Mobject and its submobjects within a bound. All of these properties may be specified by a user. I then wrote tests that would use this generator, deep copy the output, perform a specific transformation, and then assert some property held true.

Link to Generator:

https://github.com/csun-comp587-s20/manim/blob/testing/test/mobject_generator.py

Link to Property Based Tests (search for 'property'):

https://github.com/csun-comp587-s20/manim/blob/testing/test/test_mobject.py

Lessons Learned:

The original project I proposed was around 10,000,000 lines long. I would definitely count the number of lines in a SUT beforehand now. Aside from that, most of the testing I did in this project served to reinforce practices I learned in class, rather than be something completely new to me. This project also got me much more familiar with python and numpy due to its heavy usage in the SUT. If there's one thing I would improve on, it would have to be work ethic and consistency. This project would have been significantly easier and less stressful if I had put in the hours earlier, instead of at the last minute.