Timothy Beckett

Class – CSCI 3038 – Python

Project 5

For Project 5, I chose to explore the features of pygame by creating a basic shape sorting game with a reset button as a demonstration. The game's features included a reset button, along with various three-dimensional shapes like a cylinder, a cube, a prism, and a triangular trapezoidal prism. The objective was to deposit these shapes into a bucket, which added an extra challenge. Interestingly, I also included an Easter Egg – all the shapes could secretly fit into the square hole. The inspiration for this Easter Egg came from a viral video (https://www.youtube.com/watch?v=Nz8ssH7LiB0&t=59s).

Initially, I aimed to emphasize the polymorphic features of Python and extensibility in my design choices. While the resulting structures did meet this goal, it became clear that this approach wasn't the most efficient. The classes I created were complex to implement and lacked intuitiveness. Unfortunately, by the time this issue surfaced, it was too late to refactor the project. As I've said before, "Past me is an idiot."

The first major challenge arose from the fact that the game required three-dimensional shapes, which pygame doesn't inherently support. Thus, I needed to represent three-dimensional shapes within a two-dimensional space. To achieve the illusion of depth, I drew the shapes as wireframes.

In addition to the game pieces, I also needed a bucket with a lid that displayed two-dimensional shapes corresponding to their three-dimensional counterparts. Wireframes weren't sufficient here – the bucket needed to appear opaque and exhibit depth. To achieve this effect, I drew squares on a vertical plane. By arranging two vertical planes contiguously, I created the illusion of diminishing sides, thus simulating depth. I implemented a function called 'vertical_plane' which took parameters such as an ordered pair, length, slope, height, and line width. The function's calculations prioritized performance, opting for an algorithm based on trigonometric and Pythagorean Theorem concepts. This algorithm treated the length as the hypotenuse in a right triangle. The process can be broken down as follows:

1. Reorient the x and y values from the slope of the line by flipping them within the slope fraction. The length calculation then becomes y / x, representing the tangent of the angle.
2. Square the tangent to ensure a positive number and prevent incorrect results from negative inputs.
3. Assume the opposite leg of the imaginary triangle is 1, and the tangent is the adjacent leg.
4. Calculate the change in x for a given line using the square root of 1 + tangent squared.
5. Divide the length by the result from step four to obtain the line segment delta (length / square root(1 – square(y/x))).

From this point, calculating the rest of the square's perimeter using the delta and other provided pieces becomes straightforward. In hindsight, I realized that pygame's polygon function could have accomplished this task more efficiently.

The next significant challenge was enabling game piece movement while the user clicked and dragged the mouse. To address this, I needed to determine if the mouse pointer touched a game piece. Existing methods I found for this were complex. I decided to create a container class that would overlay the

game pieces, blending into the background. This container took the shape of a rectangle, and its 'collidepoint' function simplified identifying whether the mouse pointer was in contact with a game piece. However, this container class introduced some peculiar behavior when dragging game pieces to the bucket, which unfortunately went unnoticed until later stages. "It was too late in the process to address this problem" became a recurring theme in this project.