Final Write-up

Overview

We designed a customizable interface for interactive art, where the user moves their mouse to draw shapes. The user can choose between several different types of shapes (circles, squares, triangles, bowties, hexagons), and different color palettes. The user can also change the size of the shapes, or use the mirror functionality. In addition, the user can animate the shapes, making them fall, scroll, scatter, or bounce. The user can also save a screenshot of the screen to preserve their art.

Results

Figure 1: Screenshot of Program in Action

This screenshot shows a good overview of all the stationary aspects of our program. The main idea is that when the user holds down their mouse, shapes will appear wherever the user's mouse is. However, the user has quite a bit of control over the shapes that will appear by using their keyboard. The user can select which color scheme is used with the asdf keys, change size with the + and - keys, and scroll through the various types of shapes with the up and down arrows. This allows the user to explore their creativity by creating many different kinds of art.

There are also several aspects of the code that were not visible in the screenshot. The user can also save their art by pressing control-s, on which they will be prompted to type in a file name,

and the image with that file name will be saved to the same folder as the rest of the program. The user can also use control-z to undo the last few shapes drawn.

This video demonstrates the kinetic aspect of our program, such as scrolling, scattering, and bouncing, as well as the mirroring modes.
https://youtu.be/R3W2TGwgT8s
The scrolling mode moves every shape from side to side or up and down. The scattering mode picks a random speed and direction for each shape, and the shape moves in that direction. Bouncing is almost the same as scattering, except that once the shapes reach the edge of the screen, they reverse their direction, and appear to bounce on the sides of the screen. There are several mirroring modes. The user can mirror their drawing vertically, horizontally, diagonally, or both horizontally and diagonally at the same time.

Implementation

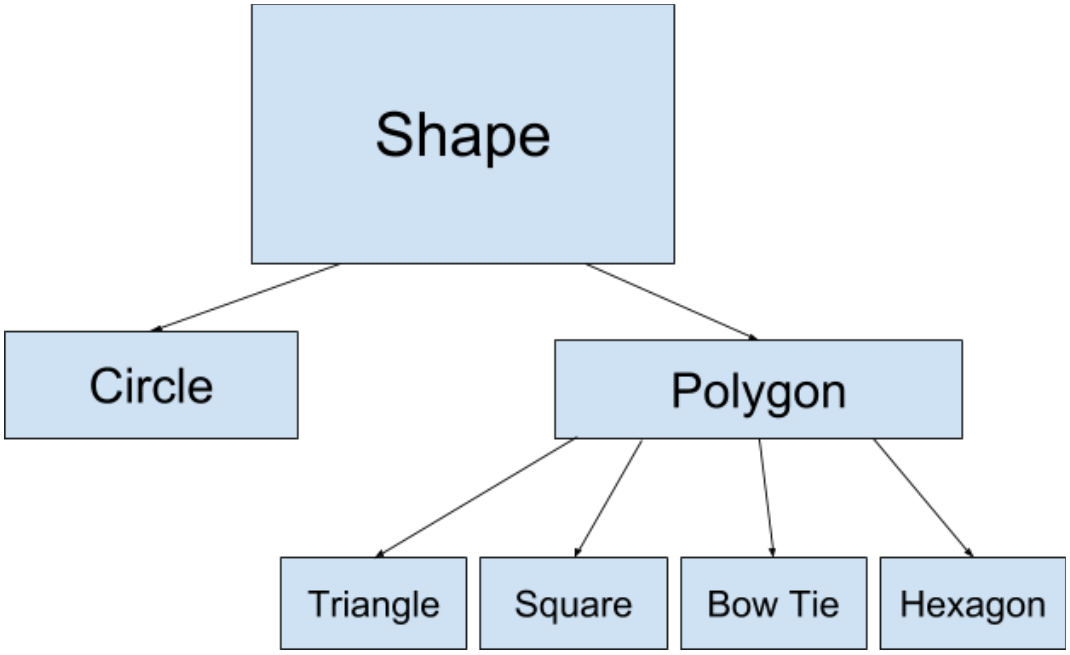

Figure 2: UML Diagram

We created a series of classes (shown in Figure 2) that encompassed the various possible shapes, taking advantage of inheritance in order to make our code more modular. Within the main program loop, we use a nested if-structure to react properly to user inputs like clicking and holding the mouse and pressing keys. This nested-if structure helps us determine the exact “settings” that the user wants for their art, like the size of the shape, the type of shape, the color, the animations, etc.

After the settings are established, we have another if-structure that reacts properly, using inheritance again to cause each shape, regardless of its specific type, to act in a certain way. Every Shape has a version of the draw method, for example, so we can easily call the draw method on an object if we know that it is a child or grandchild of the Shape class.

One design decision we had to make was the interface for inputs. We could've used buttons rather than keys, but we felt that the buttons would potentially crowd the screen and take attention away from the actual art. With the keys, however, it would be difficult to know exactly what changes you're making, which is why we output instructions informing the user how to interact with the keys. This allows the user to have maximal control over their art with the least amount of fuss.

Reflection

This project went pretty well. It was well scoped, especially since we started off with some very basic but working code, which we then built off. Our code does not contain any fruitful functions, so we were unable to use doctests. However, our code was structured in such a way that we could add on a little bit at a time, and test often. The only problem with this approach was how easy it was to get distracted by "testing" the program, and start playing with it. But nevertheless, we were able to code many working features this way.

As a team, we worked pretty well. We mostly split the work, and our only significant session of working together was in class. This approach worked well since we were both intrinsically motivated to work on this project, since it was so fun. We had a list of features to code, and whenever one of us wanted to work on the code, we would check the list to see what we still had to do, and pick one of those features to implement.