Lab 4, Part 1: Creating Shapes

The goal of this assignment is to 1) review Processing drawing properties, 2) review functions, 3) modularize code through the use of functions, and 4) introduce function parameters.

1. Review: Pre-defined Shapes
   a. It is assumed that you are familiar with the basic 2D primitive shapes (e.g. `rect`, `ellipse`, `triangle`) and their properties such as
      i. `strokeWeight`: the thickness of the line
      ii. `stroke`: the color of the foreground
      iii. `fill`: color of shape’s fill
      iv. `noFill`: removes the fill, i.e. the shape is just an outline.
      v. `noStroke`: removes the outline so that the shape is just the fill.

2. New: Polygon Shapes: A polygon shape is defined by a sequence of connected points where the last point is also connected to the first point. The connections between the points are called edges.
   a. In processing, you can create an arbitrary shape using `beginShape` and `endShape`. Look these up in the reference so that you understand the syntax.
   b. Paste into your code the example that is provided in the reference and run the program.
   c. Create your own shape using `beginShape` and `endShape`. Vary the properties as you did with the pre-defined shapes.

3. Review: Functions
   a. Functions allow you to modularize your code structure so that the code is easier to read, modify and re-use. Place your initialization code into the `setup()` function and place your polygon shape code created above into a function named whatever you like, e.g. called `makeSquareShape()`. For example:

```java
void setup() {
    size(100, 100);
    background(0, 0, 255);
    stroke(255, 0, 0);
    fill(150);
    makeSquareShape();
}

void makeSquareShape () {
    beginShape();
    vertex(20, 20);
    vertex(80, 20);
    vertex(80, 80);
    vertex(20, 80);
    endShape(CLOSE);
}
```
b. *Parameters* allow one to create variations of a shape using a single shape function. For example, run the following code. Do you understand why it generates the picture that it does?

```java
void setup() {
    size(100, 100);
    for (int i=10; i < 90; i=i+10) {
        makeRectShape(i, 10);
    }
}

void makeRectShape(int x, int y) {
    rect(x, y, 5, 80);
}
```

Try varying the parameter values of makeRectShape to see how the image changes. (Note, the current values are (i,10)).

c. Modify your code from part a) to include a loop (or nested loop!). Be ready to explain why/how the code is producing the resulting image.
d. Add several other functions that create shapes and call them from `setup()`.

---

**Lab 4, Part 2: The Translate Transformation and the Matrix Stack**

The goal of this assignment is 1) to learn about the translation transformation, 2) to continue to practice using loops, 3) to understand how transformations are applied using the modelview matrix and the matrix stack.

1. Begin by creating a single asymmetric polygon shape that fits in a small area, e.g. 50x50 pixels in the upper left corner. As in Part 1, place the code for the shape in a function (e.g. called `makeShape`) and place all other code in the `setup` function. Call `makeShape` from `setup` so that one shape object is drawn in the window, e.g. see image on right. Keep your code clean and simple.
2. The Translate Transformation: Look up the translate transformation in the Processing reference.
   a. Add a translate transformation command to your setup code. First, add it before the call to makeShape (the dots indicate unspecified other lines of code)

   ```java
   void setup() {
       translate(20,20);
       makeShape();
   }
   ```

   Run the code to see what happens. Move the translate code to after makeShape. What happens? Next, try adding several translate commands before the call to makeShape. How do the multiple translate commands compare to each individual translate? Can you replace the two translate commands with one that does the same thing?

   (Note, for animation, you can place the translate() and makeShape() in the draw() function instead of the setup() function.)

   b. Add a second call to makeShape in setup.

   ```java
   void setup() {
       makeShape();
       makeShape();
   }
   ```

   If you add a translate command before the first call to makeShape and another right before the second call to makeShape (see code snippet below), which translate(s) affect the first shape and which affect the second shape? It might help to change the stroke color so that you can tell the shapes apart.

   ```java
   translate(20,20);
   makeShape();
   stroke(255,0,0);
   translate(20,20);
   makeShape();
   ```

   What can you conclude about how the order of the transformations are applied and accumulated in Processing? Test your conclusions by trying other combinations in your code.

3. Loops and Translations:
   a. Once you are clear on how combinations of translations behave, add a loop so that you get a row of shapes as shown in the top part of the figure on the right. (You may need to increase the window
size.) Begin by asking how many shapes would fit across your window. For example, if the window is 400 pixels wide and your shape is 50 pixels wide, how many shapes would fit across the window? What if you also wanted to leave 10 pixels of blank space between the shapes as shown in the second row of the image?

b. How do you make a grid of shapes using a nested loop? For example, see image on right. This is difficult because of the way Processing accumulates the transformations. Understanding of the Matrix Stack (next part) will make this easier.

4. The ModelView Matrix: Processing keeps track of all transformations that have been encountered at each step in the code. When a shape is encountered in the code, all of the transformations that have been seen up to that point are applied to transform the shape. This may seem odd, but the transformations are applied to the shape in the reverse order in which they occur in the code. That is, the last transformation as ordered in the code (before makeShape) is the first transformation applied to the shape. For example:
   
   ```java
   translate(10,20);  // this is applied third
   translate(0,20);   // this is applied second
   translate(20, 0);  // this is applied to makeShape first
   makeShape();
   translate(20, 50);  // this does not affect makeShape.
   ```

   Note, in the above example, the order of the first three translations doesn’t really matter because translations are commutative. However, the order becomes important when we introduce the other transformations.

   One can save the sequence of encountered transformations (referred to as the current ModelView Matrix) using a function in Processing called `pushMatrix`. Later, one can retrieve this saved sequence using `popMatrix`. PushMatrix can be called multiple times to store the ModelView Matrix at various points in the code. Each time `pushMatrix` is executed, the current ModelView Matrix is placed on a stack (what is a stack?). When `popMatrix` is executed, it retrieves the ModelView Matrix that is sitting on the top of the stack. See code comments below:
pushMatrix(); // save a copy of the ModelView Matrix which at
// this point contains no transformations
translate(10,20); // add translation (T1) to the ModelView Matrix. Note,
// the saved copy still contains no transformations
translate(4,5); // add translation (T2) to the ModelView Matrix so it now
// contains T1 and T2
makeShape();  // draw shape, applying the current ModelView Matrix
// containing translation T1 and T2
popMatrix();  // retrieve saved copy of the ModelView Matrix which
// contains no transformations
translate(40,40); // add translation (T3) to the retrieved ModelView Matrix
makeShape();  // draw another shape, applying the current ModelView
// Matrix containing only the translation T3

Experiment with **pushMatrix** and **popMatrix** in your code. For example, try running the following code *with and without* the **pushMatrix** and **popMatrix** commands.

Once you understand how the ModelView Matrix and the Matrix stack work, rewrite your loop code to make use of the **pushMatrix** and **popMatrix** commands.

5. **Exercise:** (To be turned in). A design principle is to include repetition and rhythm in your picture. The repetition is most interesting when it is not exact, i.e., the repeated shape is different in shape, color, orientation. Use functions with parameters, loops, and translates to create an image that has a repeating shape (or shapes) *with variation*. That is, each time the shape is drawn, it should vary in some way either in location, color, fill, line weight, etc. Next week we will see how to easily change the size and orientation of the shape.