

Lab 3, Part 1: Practice Exercises

The goal of this part of the lab is to

- a. Review functions.
- b. Modularize code through the use of functions.
- c. Introduce function parameters.
- d. Introduce transformations: translate, rotate, and scale.

This part of the lab does not require you to turn anything in.

Section 1: Functions

1. Review of Functions

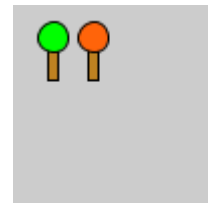
- a. Functions allow you to *modularize* your code structure so that the code is easier to read, modify and re-use. In the last lab, you placed your character code into a separate function, e.g.

```
void setup() {  
  size(500, 500);  
  background(255);  
  tree();  
}  
  
void tree () {  
  ...  
}
```



- b. *Parameters* allow one to send information to a function so as to vary the behavior of that function. For example, run the code below. When **makeTree** is called the first time with parameters (20,30,green), the value of x inside the **makeTree** function becomes 20, the value of y becomes 30, and the color becomes green, and so we get the green tree. However, when we call **makeTree** a second time, we have different values for the parameters, and so the second tree is a different color and is in a different location.

```
void setup() {  
  color green = color(0,255,0);  
  makeTree(20, 30,green);  
  color red = color(255,100, 10);  
  makeTree(40, 30,red);  
}
```



```
void makeTree(int x, int y, color leafColor) {  
  fill(190, 130, 33);      // set the color for trunk  
  rectMode(CENTER);
```

```

    rect(x, y, 5,15);           // draw the tree trunk
    fill(leafColor);           // set the color for leaves
    ellipse(x, y-14, 15,15); // draw the leaves
}

```

Try varying the parameter values to see how the image changes.

Section 2: The Translate Transformation and the Matrix Stack

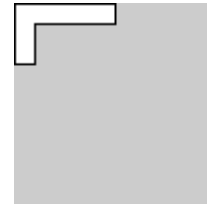
1. Begin with the code below

```

void setup() {
  size(100, 100);
  makeShape();
}

void makeShape() {
  beginShape();
  vertex(0, 0);
  vertex(50, 0);
  vertex(50, 10);
  vertex(10, 10);
  vertex(10, 30);
  vertex(0, 30);
  endShape(CLOSE);
}

```



2. Look up the **translate** transformation in the Processing reference
3. Add a **translate** transformation command to the **setup** code. First, add it *before* the call to **makeShape**:

```

void setup() {
  size(100, 100);
  translate(20,20);
  makeShape ();
}

```

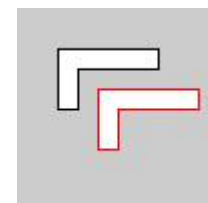
Run the code to see what happens. Move the **translate** code to *after* **tree**. What happens? Why? Next, try adding *several* **translate** commands *before* the call to **makeShape**. How do the multiple **translate** commands compare to each individual **translate**? Can you replace the two **translate**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

4. Now add a second call to **makeShape** in **setup**: 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.

```

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

```



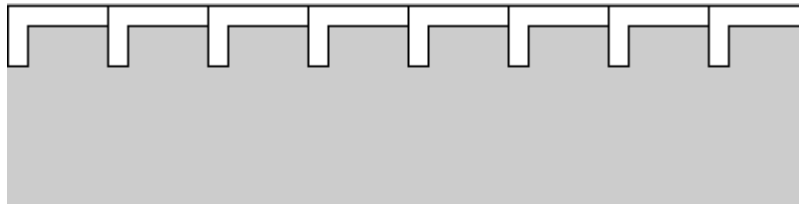
What can you conclude about how the order of the transformations are applied and accumulated in Processing? Test your conclusions by trying other combinations in your code.

5. Loops and Translations:

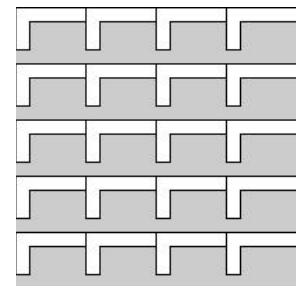
- a. Once you are clear on how combinations of translations behave, you can add a loop such as

```
void setup() {  
  size(400, 100);  
  for (int i = 0; i < width; i=i+50) {  
    makeShape();  
    translate(50, 0);  
  }  
}
```

so that you get a row of shapes as shown below.



- b. How do you make a grid of shapes using a *nested* loop as shown in the image on right. Try it – you will probably have trouble getting it to work. You will have difficulty because of the way Processing accumulates the transformations. Understanding of the *ModelView Matrix* and the *Matrix Stack* (next part) will make this easier. If you can't get the nested loop to work, don't worry, we'll come back to it later.



6. 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. And, it 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:

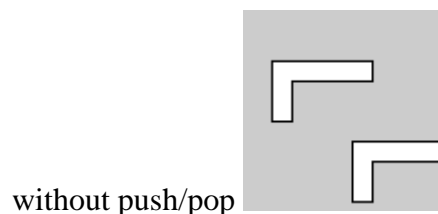
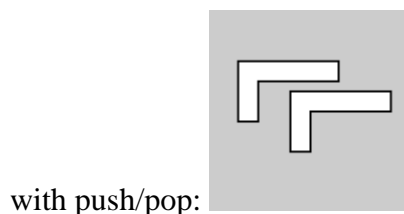
```
translate(10,20); // this is applied third  
translate(0,20); // this is applied second  
translate(20, 0); // this is applied to makeShape first  
makeShape();  
translate(20, 50); // this does not affect makeShape.
```

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 types of 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:

```
void setup() {
  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. 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 above code *with and without* the **pushMatrix** and **popMatrix**. You should get the images below. Only the second shape changes. Do you see why? And do you understand why it changed as it did?



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

Section 3: The Rotate Transformation

1. Create a new Processing sketch and paste in the following.

```
int angle = 0;
```

```

void setup() {
  size(100,100);
  background(150);
}

void draw() {
  background(150);
  rotate(radians(angle));
  makeShape();
  angle = (angle + 5) % 360; // update angle
  delay(20); // this slows down the animation
}

void makeShape() {
  beginShape();
  vertex(0, 0);
  vertex(50, 0);
  vertex(50, 10);
  vertex(10, 10);
  vertex(10, 30);
  vertex(0, 30);
  endShape(CLOSE);
}

```



Run the code to see what it does. Look up the **rotate** transformation in the Processing reference. The placement of the rotate command is very important. Try reversing the order of the **rotate** and **makeShape** to see what happens. Do you understand why the behavior changes (see the underlined note below on draw function)?

Notes:

- a. Processing requires the angle be in radians and not degrees, however, one can convert using Processing's radians function as shown above.
 - b. A *positive* angle corresponds to *clockwise* rotation in Processing.
 - c. The draw function always clears the matrix stack when draw() is called.
2. Rotations always rotate about a specific point called the pivot. The pivot is also called a fixed point because the pivot point does not move (i.e. it is fixed) when the rotation is applied. In Processing, the default pivot point for rotations is always at the origin (top left corner). In the rotate example above in part 1, you saw that the shape rotated about the origin.

One can change the pivot as follows: First try replacing draw() with:

```

void draw() {
  background(150);
  translate(width/2, height/2);
  rotate(radians(angle));
}

```

```

translate(-width/2, -height/2);
makeShape();
angle = (angle + 5) % 360; // update angle
delay(20); // this slows down the animation
}

```

Run the code. You should see the image on right:
 Here, the pivot was moved to the center, while the starting position of the shape was unchanged (upper left corner of window).



The code:

```

translate(width/2, height/2);
rotate(radians(angle));
translate(-width/2, -height/2);

```

has the effect of moving the pivot from the upper left corner of the window to the center of the window.

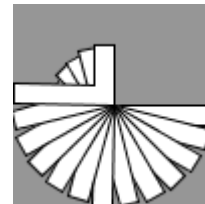
Then, try removing (or commenting out) the line:

```

translate(-width/2, -height/2);

```

Now, you will see:



The shape still rotates about center of window, but the starting location of the shape has changed. Thus, the pivot point (relative to the shape) has changed. Take your time to step through what the code is doing in order to really understand it. It will make your head hurt but once you understand the concepts, it should make sense!

3. Transformations are also useful in programs which are not animations (i.e. the ones that do not contain a draw function). Here, loops and rotations can be used to generate circular or spiral shapes. For example:

```

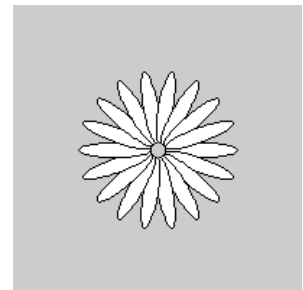
void setup() {
  size(200,200);
  translate(width/2,height/2); // translate image to center
  makeCircle();
}

```

```

void makeCircle() {
  for (int i = 0; i < 18; i++) {
    pushMatrix(); // save current matrix stack
    rotate(radians(20*i)); // rotate ellipse
    translate(30,0); // move pivot point
    ellipse(0,0,50,10); // draw ellipse
  }
}

```



```

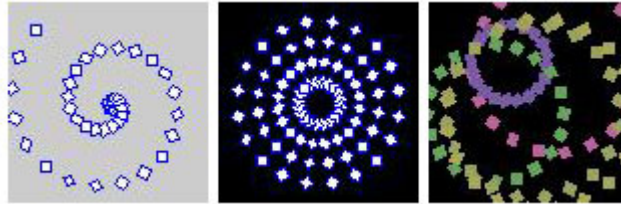
    popMatrix();          // retrieve saved matrix stack
  }
}

```

Try running the above code. Do you see why you need the push/popMatrix? Try removing them to see what happens.

Try adding additional circles at different radii composed of different shapes.

Experiment to see if you can create spirals and circles such as shown below:



Section 4: The Scale Transformation

1. Copy and paste the following code into a new Processing sketch. Run the program.

```

void setup() {
  size(100, 100);
  house();    // draw house
  scale(.5); // scale by half
  house();    // draw house again
}

```

```

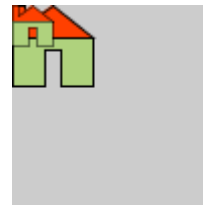
void house() {
  fill(255, 64, 0);
  beginShape(); // roof
  vertex(0, 16);
  vertex(0, 0);
  vertex(6, 0);
  vertex(6, 10);
  vertex(20, 0);
  vertex(40, 16);
  vertex(0, 16);
  endShape();
}

```

```

fill(175, 210, 125);
beginShape(); // body
vertex(0, 40);
vertex(16, 40);
vertex(16, 22);
vertex(24, 22);
vertex(24, 40);
vertex(40, 40);
}

```



```

    vertex(40,16);
    vertex(0,16);
    vertex(0,40);
    endShape();
}

```

In the above code, the house is drawn twice, once at its normal size and once at half the size. Note how the upper left corner of the house stays fixed at the origin.

We can see this more clearly in an animation. Replace `setup()` in the above code with the below. (Can you figure out what the code is doing? It is similar to the ball bouncing back and forth):

```

float scaleVal = 1.0; // scale factor
float dscaleVal = .1; // change in scaling factor

void setup() {
    size(100, 100);
    house();
}

void draw() {
    background(100);
    scale(scaleVal); // scale uniformly by amount scaleVal
    house();         // draw the house
    if (scaleVal >= 2 || scaleVal < .5) { // Change scale
        dscaleVal = -dscaleVal; // direction if scaleVal
    } // gets too small or large.
    scaleVal = scaleVal + dscaleVal; // update scale factor
    delay(50); // slow animation
}

```

2. The Scale Transformation: Look up the **scale** transformation in the Processing reference. The function `scale(scaleVal)` scales the object *uniformly* in all directions by an amount `scaleVal` relative to the origin:

When you scale a shape, observe how the position of the scaled shape changes relative to the original. In the animation above, the upper left corner stays fixed. Also note how the shape *moves away from* the origin as it grows larger and it *moves towards* the origin as the shape grows smaller. The only point that does not move is the origin. We call the origin the fixed point, which is similar to what we saw with rotations. In Processing, the default fixed point for scaling is always at the origin.

3. Non-uniform scale: One can also scale shapes *non-uniformly*, i.e. the amount of the scaling is different along one axis than the other. For example, we can scale our house by `.5*scaleVal` along the x-axis and `2*scaleVal` along the y-axis:

```

scale(.5*scaleVal, 2*scaleVal);

```

Try various non-uniform scale values.



A little later we will see how to stretch a shape along an arbitrary axis.

4. Changing the fixed point: Just as in the case of rotations, we can use translations to change the fixed point.

Note, the house is about 40x40 in size. Return the code to a uniform scale and replace the lines

```
scale(scaleVal);  
house();
```

with the lines

```
translate(20, 20);  
scale(scaleVal);  
translate(-20, -20);  
house();
```



When you run the code, the fixed point is at the center of the house instead of the upper left corner. How would you change the code so the fixed point is at the lower left corner of the house?

Why is this useful?: If the house is positioned on “the ground” and the fixed point is anywhere along the base of the house then the base will stay fixed to the ground no matter how we scale. This is desirable because we may want the house to stay level on the ground even if the house is resized. In the image below, each house was obtained by setting the fixed point to be at the base of the house and then scaling non-uniformly and translating horizontally. *No vertical adjustment was needed.* This would not have been the case if the fixed point had not been at the base of the house. In general, it is important to think about where you want your fixed point to be because a good choice can greatly simplify drawing images.

