CS 145 Images and Imagination, Fall 2010

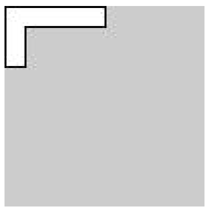
Lab 6, Part 1: Practice Exercises

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.

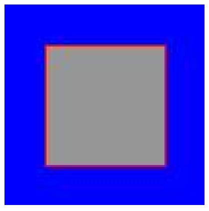
Section 1: Functions

1. 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. An example is shown below within the function makeSquareShape().
    b. Create your own shape using **beginShape** and **endShape**. Vary the properties as you did with the pre-defined shapes. Make sure that your shape:
        i. Is small, e.g. fills a region of space no larger than, say, 60,60.
        ii. Have the shape be somewhat asymmetric so that if it is rotated, it will look clearly different. See example to the right.
        iii. Is created near the origin and not, for example, in the middle of the window.



2. Review of Functions
    a. Functions allow you to *modularize* your code structure so that the code is easier to read, modify and re-use. Place the shape you created above into a function as shown below for `makeSquareShape()`. You may call your function anything you like, however, it is good practice to give it a name that reflects what it is drawing.

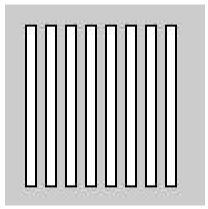
```
void setup() {
  size(100, 100);
  background(0,0,255);
  stroke(255,0,0);
  fill(150);
  makeSquareShape();
}

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. When makeRectShape is called with parameters (i,10), the value of i gets copied into the variable x and the value 10 is copied into the variable y.

```
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. For example, what happens if you call makeRectShape(i,i) instead of makeRectShape(i,10)?

Section 2: The Translate Transformation and the Matrix Stack

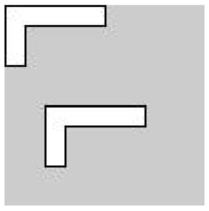
1. Create a simple Processing program as shown in part 2 (Review of Functions) above. Use your own shape function. For the purpose of illustration, we will assume that your shape function is called **makeShape()**.
2. Look up the **translate** transformation in the Processing reference
3. Add a **translate** transformation command to your **setup** code from above. First, add it *before* the call to **makeShape** (the dots indicate unspecified other lines of code)

```
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?

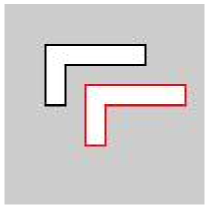
4. Add a second call to **makeShape** in **setup**.

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



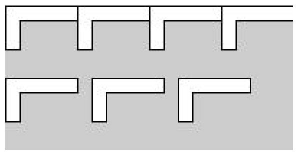
If you add a **translate** command before the first call to **makeShape** and another right before the second call to **makeShape** (see code snippet below), which translate(s) affect the first shape and which affect the second shape? It might help to change the stroke color so that you can tell the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

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

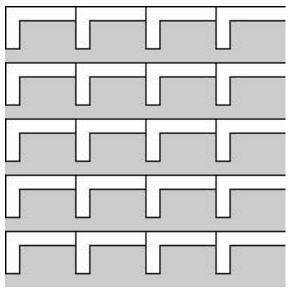


What can you conclude about how the order of the transformations are applied and accumulated in Processing? Test your conclusions by trying other combinations in your code.

5. Loops and Translations:
   a. Once you are clear on how combinations of translations behave, 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.



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,

```
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
```

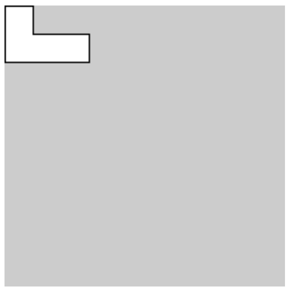
Experiment with **pushMatrix** and **popMatrix** in your code. For example, try running the following code *with and without* the **pushMatrix** and **popMatrix**.

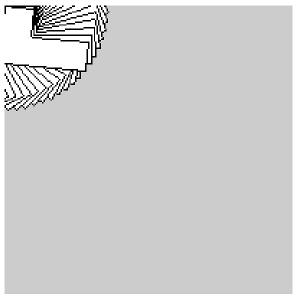
Once you understand how the ModelView Matrix and the Matrix stack work, rewrite your nested loop code to make use of the **pushMatrix** and **popMatrix** commands.

Section 3: The Rotate Transformation

1. Create a new Processing sketch and paste in the following. You may replace myShape with the shape function you created above.

```
int angle = 0;

void setup() {
  size(400,400);
  background(0,0,255);
}

void draw() {
  background(0,0,255);

  rotate(radians(angle));
  myShape();

  angle = (angle + 5) % 360; // update angle
  delay(20);  // this slows down the animation
}

void myShape() {
  beginShape();
  vertex(0, 0);
  vertex(20, 0);
  vertex(20, 20);
  vertex(60, 20);
  vertex(60, 40);
  vertex(0, 40);
  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 **myShape** to see what happens. Do you understand why the behavior changes?

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 the function is called.

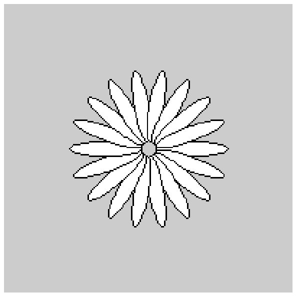
2. Rotations always rotate about a specific point called the pivot. The pivot is also called a fixed point because the pivot point does not move (i.e. it is fixed) when the rotation is applied. In Processing, the default pivot point for rotations is always at the origin (top left corner). In the above example, you should see the object rotate about the origin.

Given the code

```
translate(width/2,height/2); // move pivot to window center
rotate(angle);      // rotate the shape
translate(-x,-y) ;  // move the pivot point at (x,y)
                    //          to the origin
myShape();          // draw the shape
```

The **translate** immediately before **myShape** changes the pivot *relative to the shape*. The **translate** before the **rotate** changes the location of the pivot *in the window*.

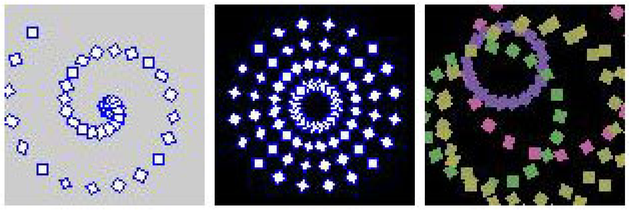
3. Transformations are also useful in programs which are not animations (i.e. the ones that do not contain a draw function). Here, loops and rotations can be used to generate circular or spiral shapes. For example:



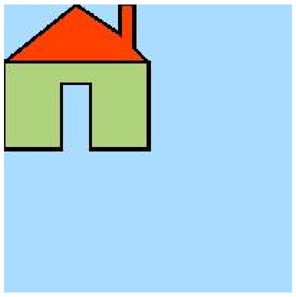
```
void setup() {
  size(200,200);
  translate(width/2,height/2);  // translate image to center
  makeCircle();
}

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:



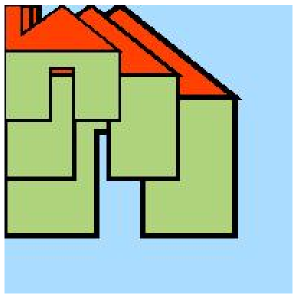
Section 4: The Scale Transformation



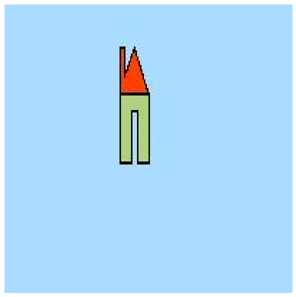
1. Copy and paste the following code into a new Processing sketch, replacing myShape with your own shape. Run the program.

```
void setup() {
  size(400,400);
}

float scaleVal = 1.0;

void draw() {
  background(100);
  scale(scaleVal);    // scale uniformly by amount scaleVal
  myShape();                         // draw the shape

  if (scaleVal >= 3) scaleVal = .2; // reset when gets to 3
  scaleVal += .1;                    // update scaleVal
  delay(50);                         // slow animation
}

void myShape() {
    // add your own code here
}
```

2. The Scale Transformation: Look up the **scale** transformation in the Processing reference. In the above code, the 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. Specifically, 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.



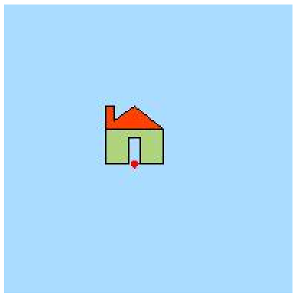
3. Non-uniform scale: One can also scale shapes *non-uniformly*, i.e. the amount of the scaling is different along one axis than the other. For example, we can scale our house by scaleVal along the x-axis and 2*scaleVal along the y-axis:

```
        scale(scaleVal, 2*scaleVal);
```

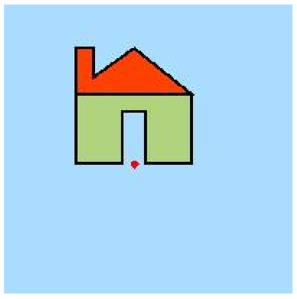
   Try various non-uniform scales with your shape.

   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 use translations to set the fixed point as well as the location of the shape in the window.

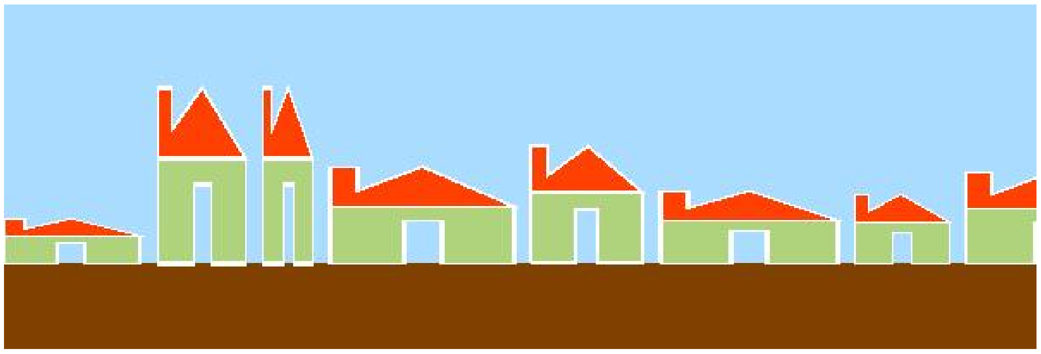
   For example, suppose our house is sitting as shown in the image to the right and suppose that we want to scale the house *about the red dot* (this is the pivot) located at position (90, 110).

   Then the code would be

```
     translate(90,110);     // translate back
     scale(2);              // scale
     translate(-90,-110);   // translate fixed
                     //  point to the origin
     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 (as in the above example) 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 scaling non-uniformly and translating horizontally, however, *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.



5. Remember: Write clean and easy to manage code:
   a. Identify the different elements in your image. For example, you might have a house, the ground, and the sky as shown above.
   b. Separate each element into a separate function. Make sure that each function has as few side-effects as possible. For example, if you change the fill color in your function, make sure at the end of the function you change it back to what it was originally.
   c. To draw the multiple elements, you can call the functions separately. You may need to reset or push/pop the matrix stack to insure that each layer behaves independently.
   d. Use variable names that identify what the variable represents.