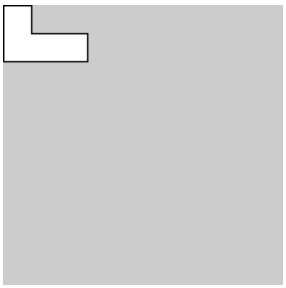
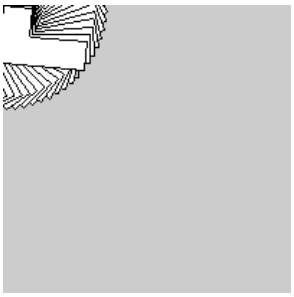
Lab 5, Part 1: The Scale and Rotate Transformations

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

```
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.

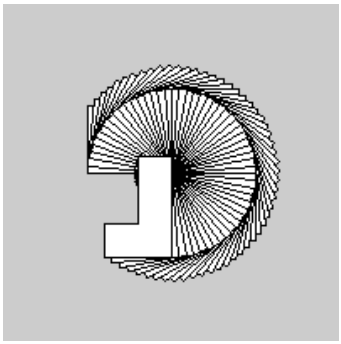
2. Rotations always rotate about a specific point called the pivot. The pivot is also called a fixed point because the pivot point does not move (i.e. it is fixed) when the rotation is applied. In Processing, the default pivot point for rotations is always at the origin (top left corner). In the above 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. Replace myShape() with a shape placed near the origin which you have created. The shape should not be large so that you can more easily see what is happening with the rotation.



4. Add a **translate** transformation to the line right *before* the **rotate**. What happens?
   Add a **translate** transformation to the line right before the **rotate**. What happens?

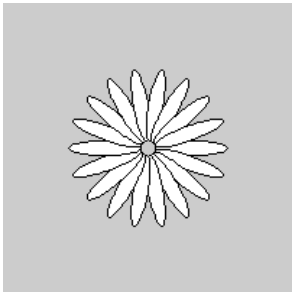
   Experiment by changing the pivot and location of the pivot in the window. If someone gives you a pivot point and window location, you should know exactly what translations are needed to draw an image with this pivot and location.

5. It is important that you can explain the above behavior in terms of the matrix stack. Use pencil and paper to work through what the matrix stack looks like at each step in the code. Specifically what does the stack look like at the point when myShape is executed?

6. As you can see above, rotation and translation are not commutative with each other, that is, the order in which they appear in the code matters!

   However, in Lab 4, we saw that two translations, *which are adjacent in the code*, are commutative (if you reverse their order, nothing changes). Are two *adjacent* rotations commutative with each other? Try it out to find out. Why or why not?

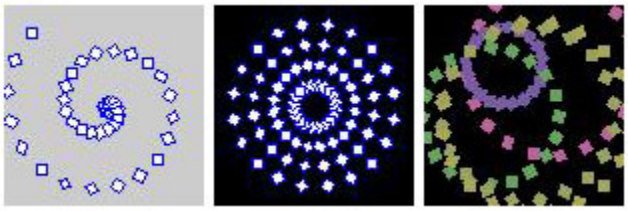
7. Transformations are also useful in programs which are not animations (i.e. the ones that do not contain a draw function). Here, loops and rotations can be used to generate circular or spiral shapes. For example:



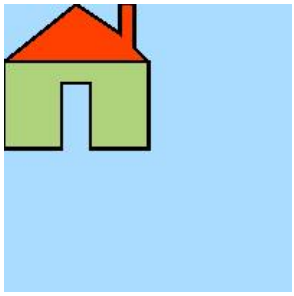
```
void setup() {
  size(200,200);
  translate(width/2,height/2);  // translate image to center
  makeCircle();
}

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.

8. **Exercise 1:** Circles and spiral shapes are useful from a design perspective because they can focus where the viewer looks. Create a new Processing program. Use loops, rotation and translation transformations to obtain 1) a spiral shape, 2) a sequence of circles of increasing radii, 3) random circles as shown below. Save several examples to image files.



Lab 5, Part 2: 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
}
```

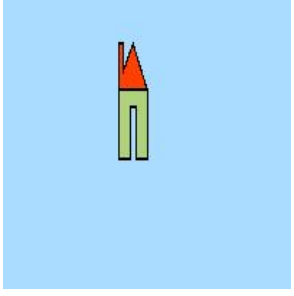
2. The Scale Transformation: Look up the **scale** transformation in the Processing reference. In the above code, the scale(scaleVal) scales the object *uniformly* in all directions by an amount scaleVal relative to the origin:

   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.



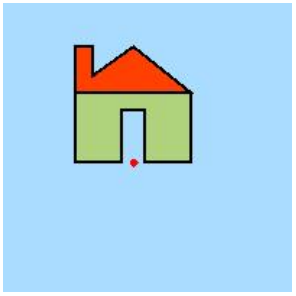
A little later we will see how to stretch a shape along an arbitrary axis.

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();
```



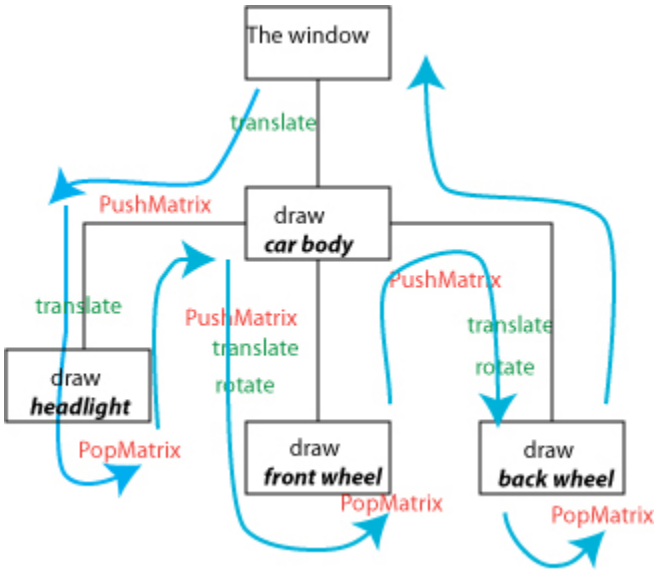
Why is this useful?: If the house is positioned on "the ground" and the fixed point is anywhere along the base of the house (as in the above example) then the base will stay fixed to the ground no matter how we scale. This is desirable because we may want the house to stay level on the ground even if 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. **Exercise 2:** For the shape you have created, pick a fixed point at the base and create a row of shapes randomly sized but which are level horizontally as the houses are above. Save a copy of the image.

6. Remember: Write clean and easy to manage code:
    a. Identify the different “layers” or elements in your image.
    b. Separate each layer into a separate function. Make sure that each function has as few side-effects as possible.
    c. To draw the multiple layers, you will call the functions separately. You may need to reset or push/pop the matrix stack to insure that each layer behaves independently.
    d. Place closing brackets on a line by themselves.
    e. Use the auto format to insure proper indentation.
    f. Add comments if it isn’t obvious what the code does.
    g. Use variable names that identify what the variable represents.

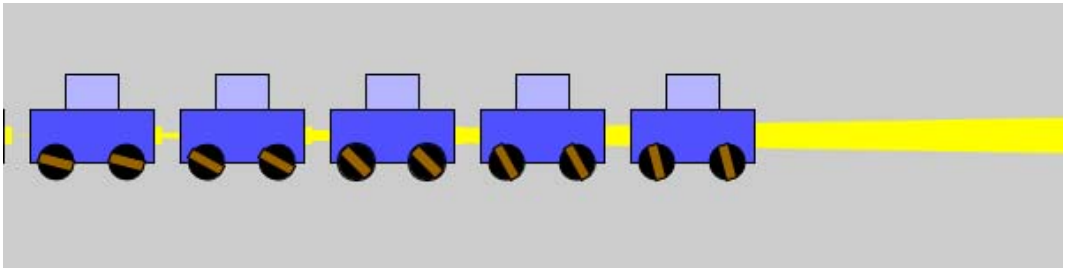
Lab 5, Part 3: Hierarchical Structures

Exercise 3: In class, we looked at hierarchical structures (e.g. solar system and robot arm). Pick a simple hierarchical system (e.g. a car with wheels where the wheels turn as the car moves across the screen). Draw a graph of the structure with paper and pencil to identify what transformations are needed. For example:



Once you have done this, build the structure in your code using functions for each part and push/popMatrix to maintain the proper transformation structure. Generate a single image snapshot (using save()) and also generate an applet.

A single snapshot



The car at a sequence of times moving right. Note, the rotating tires.