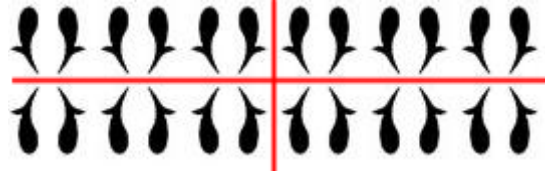


Lab 3, Part 2: Symmetry: Frieze Patterns

Symmetry refers to the ways in which a pattern or image repeats itself. More precisely, we say a design exhibits symmetry if it can be transformed (a combination of rotate, translate, mirror) in a way such that the transformed image sits exactly on top of the original image. For example, the following image does not change if you mirror it in either the horizontal or vertical red lines.



Symmetric objects have an esthetic appeal and thus occur frequently in art and architecture. However, nature also loves symmetry as can be seen in objects such as flowers, starfish, and crystals, to name a few. The human body is symmetric about its center, front vertical axis.

One of the simplest types of symmetric patterns to understand are frieze patterns which are linear patterns often used as borders in paintings or architecture.



Figure 1 All Saints Chapel in St Louis Cathedral Basilica

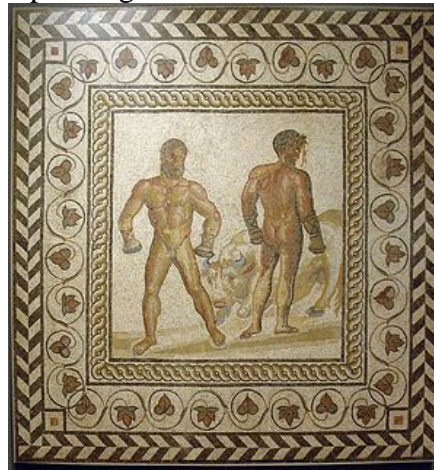


Figure 2 From the Getty Villa in LA

We can think of a frieze pattern as an infinitely long repeating pattern. Quite surprisingly, it can be proven (we won't prove it here) that there are only 7 types of frieze symmetries, as shown in the Figure 3. That is, given a linear design that exhibits some type of symmetry, there are only 7 potential ways of transforming it so that the design remains unchanged!

The names, e.g. “hop”, were fancifully coined by mathematician John Conway based on what a person would have to do to generate a symmetric foot print pattern. For example, to generate the hop pattern, someone would stand on one foot starting at the left and hop by some fixed amount to the right. Thus the hop pattern is *generated* by translations (by

some fixed amount) to the right of some base image (also called the lattice shape, motif, or icon), in this case, a footprint. Thinking of it another way, if we translate the entire hop frieze pattern to the right (or left) by the width of the lattice, the pattern will remain unchanged.

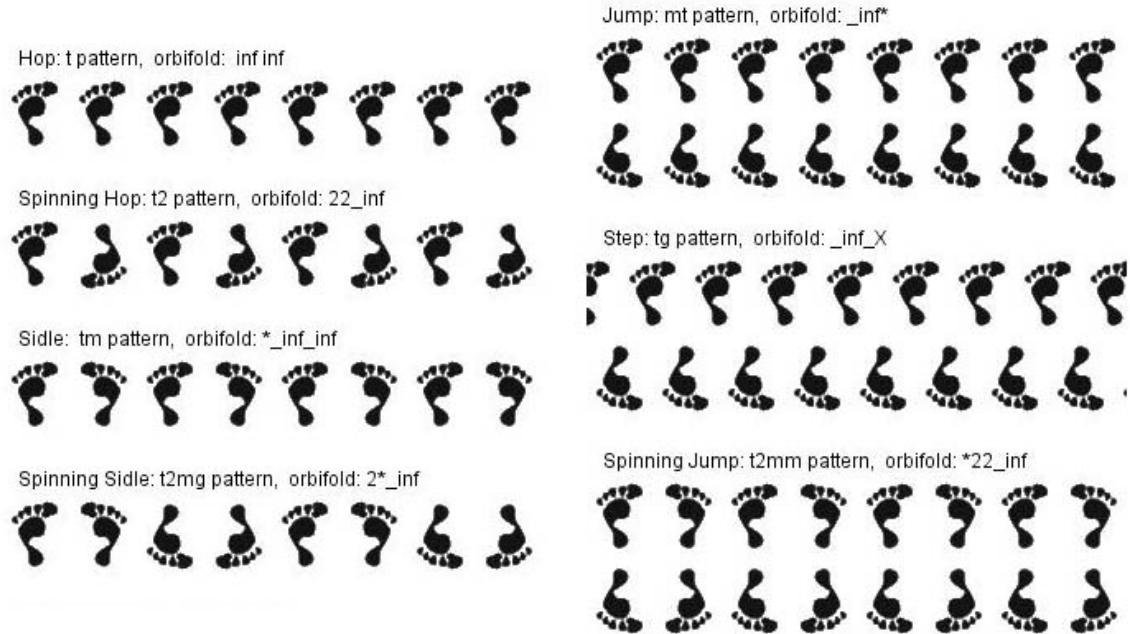


Figure 3 The Seven Frieze Patterns

Figure 4 shows several examples of the “spinning hop” symmetry. While the lattice is different in each case, the symmetry structure is the same, i.e. rotation by 180 degrees about red dots (e.g. as seen on right image).



Examples of the Spinning Hop symmetry

Figure 4

We will use Processing, together with our understanding of transformations, to generate each of the frieze symmetry patterns. To do this, we must identify the underlying transformations associated with each symmetry pattern.

We begin with the simplest, the hop pattern, where the transformation is simply a horizontal translation. We will go through the code in detail for several of the frieze patterns. The assignment is to implement the remaining patterns based on the same process.

The two most important programming rules to follow are

1. Use very simple test problems that are easy to check.
2. Break the problem into very small steps. Implement and thoroughly test each step before continuing to the next step.

If you follow these rules, everything will end up being easy!

Creating the Lattice or Base Image

Before we begin, we need to create several lattice shape images. Here are recommendations to follow in selecting a lattice image:

1. The image should be scaled so that it is small, e.g. 50-75 pixels on a side.
2. The image can be anything you want but you should always *start with something simple* like the images used in the patterns Figure 5, a-d below. If you begin with a complex pattern like e or f, it will be difficult to tell if you have generated the correct transformation. Once you know your code works, you can run your code with other lattices images. In fact, it can be fun and surprising to see the friezes generated with a range of more complex lattice images.

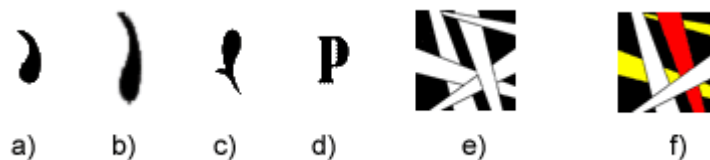


Figure 5

3. Use an *asymmetric* shape, e.g. like the letter “P”. The letter “A” is not good because it is symmetric about its center axis and so it will be difficult to see certain symmetry patterns. Again, refer to in Figure 5a-d for examples.
4. There will be a folder placed on our shared drive containing lattice images you can use. You can also use the small image of your character from lab 1, as long as it isn’t symmetric or complicated (once you have your programs tested, you can run it with your character image). You may also use Paint or Photoshop to create an image.

5. Consider images with different dimensions, e.g. square (height=width) or elongated (e.g. height = 2*width). Some symmetric patterns work best with certain dimensions and some work well with any dimensions. You will discover these characteristics as the assignment is completed.

The Hop Pattern

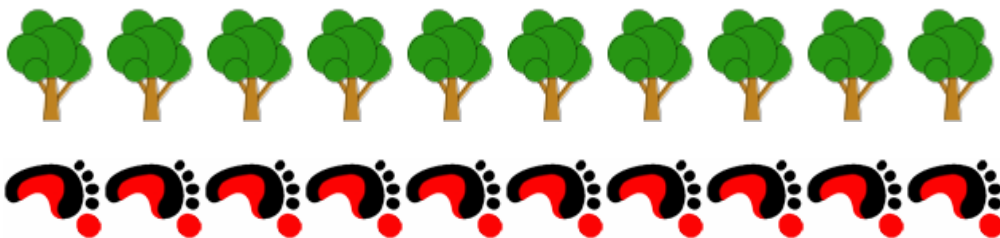
Here we discuss how to obtain a Hop in a methodical way so that it can easily be extended later to obtain more complex symmetry patterns.

1. Begin a new Processing program and place your lattice image into its sketch folder.
2. Copy the code below into the Processing code window (it may be easier to copy the code from [here](#)) You will need to change the `icon.jpg` to the name of your lattice image. This program generates a hop frieze pattern.

```
1  PImage icon;    // storage for image
2  int w, h;       // width and height of image
3  int reps = 10;  // number of repetitions of
                  // image across window

4  void setup() {
5    icon = loadImage("icon.jpg");
6    w = icon.width;
7    h = icon.height;
8    size(reps*w, h); // set window size
9    drawFrieze();
10   save("hopFrieze.gif");
    }

// Draw Hop frieze pattern
11 void drawFrieze() {
12   for (int i = 0; i < reps; i++) {
13     image(icon, 0, 0);
14     translate(w,0);
    }
```



3. Comments on the code:
 - a. We have structured the code using functions. It is important to do this because it keeps the code clean, simple and very readable. It also will greatly simplify later work.

- b. *It is good programming practice to anticipate code modifications and to write the code so that any modifications will require changing as few lines of code possible.* For example:
 - i. Creating variables `w` and `h` (lines 2, 6, 7) was not necessary since we could have just kept using `icon.width` and `icon.height`. However, these two variables will be used a lot in later code. The code is more readable and more easily modified if you use the variables `h` and `w`.
 - ii. The window size is set in terms of `w`, `h`, and `reps` (line 8). This way, if you replace the current lattice image with a new lattice image of different size, then you only need to change the file name since the window size will automatically adjust.
 - iii. By creating the variable called `reps` (i.e. short for repetitions), you can easily increase the number of lattice images that are drawn. Note that the variable `reps`, which is the number of lattice images drawn, is used in lines 8 and 12. We could have just used the number 10 directly, however, this would have made it more difficult to change the repetitions to another value.
- c. The hop frieze pattern is generated by a simple translation. Carefully examine the loop in the function `drawFrieze()` to understand what it is doing.

The Spinning Hop Pattern



The hop pattern is fairly simple to generate. To generate a more complex pattern requires carefully analyzing and understanding the sequence of transformations needed to generate the pattern. Consider the spinning hop pattern.



Figure 6

One can rotate about either the red or green points. If we consider rotating the foot image about one of the red points, we get the pair of feet as outlined by each of the blue rectangles. We will call this the *fundamental region*. If we take this fundamental region and translate it (just as in the hop pattern) then we get the spinning hop.

Thus we can generate the pattern by following the steps (*these will be elaborated on later!*):

1. Generate a rotated foot: 
2. Combine with the original foot to get the fundamental region: 

3. Repeatedly translate, to get the frieze pattern:



Once the above steps are identified, the implementation should follow the process:

- *Implement each of the above steps as separate functions.* The function for step 2 will then call the function for step 1, and the function for step 3 will call step 2.
- *A pushMatrix and popMatrix should always be the first and last line of each function* to insure that there are no unexpected side effects from calling the function. That is, we want the matrix stack at the beginning of the function to be the same as the matrix stack when the function ends.
- *Stepwise Refinement:* Implement the functions one at a time, starting with the simpler ones (e.g. step 1). Thoroughly test each function to see that it works *before* moving on to the next function.

Elaboration of Steps above:

Consider step 1: If we execute the line:

```
image(icon, 0, 0);
```

we draw our image at the upper left corner of the window:



Figure 7

Our first goal is to determine the transformations needed to get **Figure 7** to **Figure 8**:



Figure 8

To do this, we need to do a 180° rotation about the *center* of the foot image. Recall from part 1 that, to rotate about a pivot point, we 1) translate that point to the origin, rotate, and translate back. These actions are implemented in the code in the reverse order.

Therefore, we have the following function

```
void rotateHalfTurn() {  
    pushMatrix();           // Save the current Matrix Stack  
    translate(w/2,h/2);     // translate back  
    rotate( radians(180)); // rotate by a half turn  
    translate(-w/2,-h/2);  // translate center to origin  
    image(icon, 0, 0);     // draw image  
    popMatrix();           // Retrieve the saved Matrix Stack  
}
```

Note, there are other ways of achieving the same result. However, the above process will consistently work under many circumstances and so is useful to use and understand.

We can test this code by modifying our hop pattern code

```

PImage icon;    // storage for image
int w, h;      // width and height of image
int reps = 10; // number of repetitions of
               // image across window

void setup() {
  icon = loadImage("icon.jpg");
  w = icon.width;
  h = icon.height;
  size(reps*w, h); // set window size
  rotateHalfTurn(); // test your function
  save("rotateHalfTurn.gif");
}

// insert function rotateHalfTurn here

```

You should test to see that your code is working before continuing!

The result should be what you see in **Figure 8**. Note, once we know that the above function works, we no longer have to think about how to rotate the footprint; we just call the function instead.

Consider Step 2: Now that step 1 is done, it is simple to do our next step which consists of the rotated image placed to the right of the original image. The function for step 2 reduces to a simple translation:

```

void basePattern() {
  pushMatrix();
  image(icon, 0, 0); // draw upright foot
  translate(w,0);    // translate rotated foot
  rotateHalfTurn(); // draw rotated foot
  popMatrix();
}

```

Test to see that your code is working before continuing!

The result should be our fundamental region (base pattern):



Figure 9

Consider Step 3: We now just need to modify the drawFrieze function from our hop pattern code. There are two changes. First, we need is to translate by twice the image width since the width of step 2 is twice the width of the original image:


```

void drawFrieze() {
    for (int i = 0; i < reps; i++) {
        basePattern ();
        translate(2*w,0);
    }
}

```

Second, we need to increase the window width by changing `size(reps*w, h)` to `size(2*reps*w, h)` because each repetition is now twice the width of the original image. The result is:



Figure 10

Your Assignment

Implement at least 3 of the remaining 5 frieze patterns shown in **Figure 3**: (sidle, spinning sidle, jump, step, spinning jump). Follow a process similar to what was done for the spinning hop:

- a. analyze the sequence of steps needed to generate the frieze
- b. for each step, write a function that implements that step
- c. test that your code works at each step before moving to the next step.

Once you have a sketch for each of the 3 frieze patterns, run each of the sketches with at least 2 different lattice images (one simple asymmetric shape, and one of your choosing) so that you have *at least 2 different friezes for each sketch*.

See the [main Lab 3 instruction page](#) for how to submit your work.