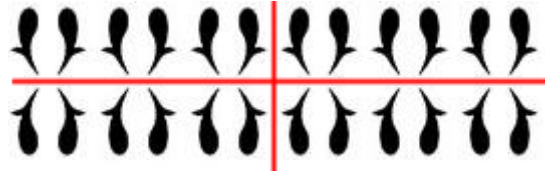


Lab 6: Symmetry: Frieze Patterns

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 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

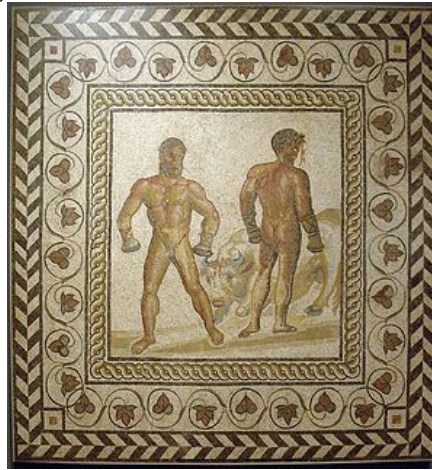


Figure 2 From the Getty Villa in LA

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 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, an image of a foot print. 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.

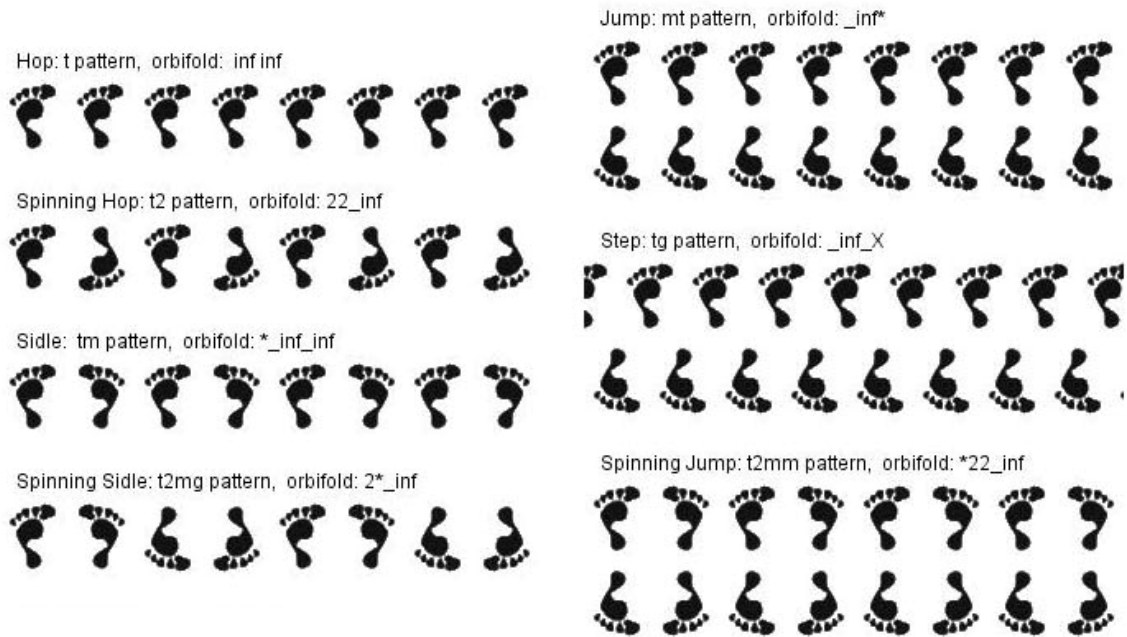


Figure 3 The Seven Frieze Patterns

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).



Red dots show centers of rotation.

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.

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. The image should be scaled so that it is small, e.g. 50-75 pixels on a side. The image can be anything you want but you should always *start with something simple* like the image used in the patterns Figure 5, a)-d) below. If you begin with a complex pattern like e), it will be difficult to tell if you have generated the correct transformation.

To create the lattice shape, you may use a shape function created in Processing as in previous assignments, or you may use Paint or Photoshop.

Suggestions:

1. Use a simple *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. Below are some examples. Do not *start* with pattern e) or f) because they are too complex. However, once you know your code works, it can be fun and surprising to see the friezes generated with a range of more complex lattice images.

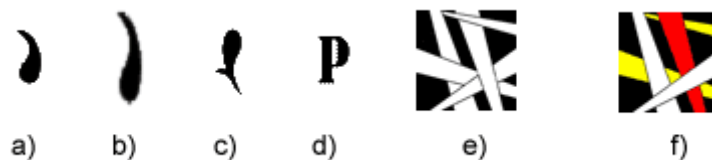


Figure 5

2. For some of the symmetry patterns, particularly those that require rotations, one can get overlap between transformed images. For this reason, it is recommended that the image have a transparent background. In this case, one must use the gif or png format since jpg doesn't support transparency.
3. Include images with different dimensions, e.g. square (height=width) or elongated (e.g. height = 2*width). Some symmetric patterns only work with certain

dimensions and some work with any dimensions. You will discover these characteristic as the assignment is completed.

The Hop Pattern

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 generates a hop frieze pattern. Remove the line numbers. You will also need to change the `icon.jpg` to the name of your lattice image.

```
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);
    }
}
```

Note, if you create a Processing shape drawn in a function instead of using an image, you will need to replace line 13 with your function call.

3. Open up the Processing reference and read about the datatype `PImage` and the functions: `setup()`, `loadImage()`, `image()`, and `save()`.
4. 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. Recall that the `setup()` function, beginning in line 4, is a special function in Processing that gets called once in the very beginning when the program is executed.

- c. *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`, 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.
- d. The hop frieze pattern is generated by a simple translation. Carefully examine the loop in the function `drawFrieze()` to understand what it is doing.

Variations (Optional)

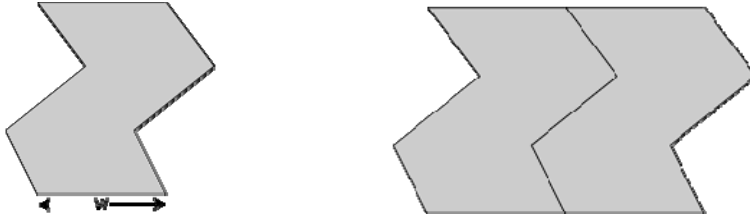
Note that the lattice image does not have to be rectangular for the hop pattern. The only requirement is that the left edge fits into the right edge. For example, we could use the rhombus shape shown at the right to obtain:



The image itself as created in, say, Photoshop, must be rectangular. Therefore, to create the rhombus lattice, one creates a rectangular image as shown by the black outline to the right. The non-gray areas are made transparent so they do not appear in the final image. *The size of the translation is determined by the width of the gray area, and not the width of the image as defined by the black outline.*



Since the only requirement is that the left and right sides fit together like puzzle pieces, we could even use a shape such as



However, again, we must be careful to translate by the width of the gray at the base of the image rather than the actual width of the rectangular image.

Optional: Use Photoshop (or some similar program) to create a non-rectangular lattice image. Modify your code to generate the hop frieze pattern so that it uses the proper width.

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 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 (this will be elaborated on later):

1. Generate a rotated foot: 
2. Combine with the original foot to get: 
3. Repeatedly translate, to get the frieze pattern: 

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

1. *Implement each of the above steps as a separate function.* Obviously, the function for step 2 will use the function for step 1, and step 3 will use step 2.

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.
3. Start with the simpler functions (e.g. step 1) and test that it works before moving on to the next step.

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

Thus, our first goal is to determine the transformations needed to get Figure 7 to Figure 8:



Figure 8

Recall from previous assignments 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();
    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();
}
```

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.

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();
    save("rotateHalfTurn.gif");
}

// insert function rotateHalfTurn here
```

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

The result is **Figure 8** above. 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.

Therefore, it is simple to do our next step which consists of the rotated image 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



Figure 8

For step 3, we now just need to modify the drawFrieze function from our hop pattern code:

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

Note, we needed to translate by twice the image width since the width of step 2 is twice the width of the original image.

The result is:



Figure 9

Exercise: Implementing any of our frieze patterns follows the exact same process. Your job now is to implement at least 3 of the remaining 5 frieze patterns shown in **Figure 3:** (sidle, spinning sidle, jump, step, spinning jump). Use the process described above:

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

Once you have the code written for each frieze pattern, generate the frieze pattern for at least 2 different lattice shapes, one of which should be a simple asymmetric shape.