Design Programming

DECO1012

Rob Saunders
Rob Saunders

web: http://web.arch.usyd.edu.au/~rob

e-mail: rob.saunders@sydney.edu.au

office: Room 274, Wilkinson Building
Arrays, Pixels & Motion
Arrays
Data-Structures

Data-structure is a general term used to describe different ways of organising data in a computer’s memory so that they can be accessed or manipulated easily.

Some examples of common data-structures include arrays, lists, sets, maps and trees.

We’re only going to look at the most common type: arrays
Arrays

An array is a way to store multiple items of data in memory so that they are accessed using a single variable.

Each item of data in an array is given an index that can be used to refer to it.

Arrays can be used whenever a program needs to store multiple copies of same type of data in one place.
An Array of Characters

Index | 0 1 2 3 4 5 6 7 8 9
Data   | PROCESSING
Declaring an Array

Declaring a variable that points to an array is similar to declaring variables that point to primitive data types.

The type of the data to be stored in the array is simply followed by square brackets.

```
int[] array0fInts;
```
Allocating an Array

Before an array can be used the memory that it needs to store all the data has to be allocated. Memory is allocated using the “new” operator.

To allocate memory for an array the new operator needs to know the type of number of data items that will be stored.
Allocating an Array

An array is allocated by following the new operator by the type of data followed by the size of the array in square brackets:

\[
\text{arrayOfInts} = \text{new int}[42];
\]

It is also possible to use integer variable types to calculate the size of an array:

\[
\text{arrayOfInts} = \text{new int}[x + 5];
\]
Declaring and Allocating Arrays

It is also possible to declare and allocate an array in a single statement:

```java
int[] arrayOfInts = new int[42];
```

Unlike primitive data types however, this does not assign the values of the array. The program has to do this later by setting the value of each data item.
Assigning Values to Arrays

To access an item we use the variable name followed by the index in square brackets.

```java
char[] vowels = new char[5];
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
```

Notice how the indices start at 0. This array with five items has indices 0...4.
Assigning Values to Arrays

It is possible to assign the values stored in an array don’t need to change using a special type of array allocation:

```java
int[] arrayOfPrimes = {1, 2, 3, 5, 7};
```

This allocates an array big enough to hold five integers and initialises their values to the first five prime numbers.
Declaring, Creating and Assigning Arrays

```java
int[] data; // Declare
data = new int[5]; // Create
data[0] = 19; // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```

```java
int[] data = new int[5]; // Declare and create
data[0] = 19; // Assign
data[1] = 40;
data[2] = 75;
data[3] = 76;
data[4] = 90;
```

```java
// Declare, create, and assign
int[] data = { 19, 40, 75, 76, 90 };`
Array Index Out Of Bounds Exception

- If you try to access a member of the array that lies outside the array boundaries, your program will terminate and give an `ArrayIndexOutOfBoundsException`.

```java
int[] data = {19, 40, 75, 76, 90};
println(data[0]);  // Prints 19 to the console
println(data[2]);  // Prints 75 to the console
println(data[5]);  // ERROR! The last element of the array is 4
```
Array Length

- The `length` field stores the number of elements in an array.

```java
int[] data1 = { 19, 40, 75, 76, 90 };
int[] data2 = { 19, 40 };
int[] data3 = new int[127];
println(data1.length); // Prints "5" to the console
println(data2.length); // Prints "2" to the console
println(data3.length); // Prints "127" to the console
```
Using **for** Loops with Arrays

- A **for** loop can be used to access array elements, especially with large arrays.
- The following example uses a **for** structure to iterate through every value in an array.

```java
int[] data = {19, 40, 75, 76, 90};
for (int i = 0; i < data.length; i++) {
    line(data[i], 0, data[i], 100);
}
```
Using **for** Loops with Arrays

```java
float[] sineWave = new float[width];

for (int i = 0; i < width; i++) {
  // Fill the array with values from sin()
  float r = map(i, 0, width, 0, TWO_PI);
  sineWave[i] = abs(sin(r));
}

for (int i = 0; i < sineWave.length; i++) {
  // Set the stroke values to numbers read from the array
  stroke(sineWave[i] * 255);
  line(i, 0, i, height);
}
```
Using Arrays to Draw

- Arrays can make the task of programming much easier.
- While it’s not necessary to use them, they can be valuable structures for managing data.
- Arrays can be created to hold any type of data, and each element can be individually assigned and read.
Using Arrays to Draw

```
(50,18)  (39,37)  (61,37)
(17,43)  (50,73)  (83,43)
(31,60)  (50,73)  (69,60)
(29,82)  (71,82)
```
Separate variables

```
int x0 = 50;
int y0 = 18;
int x1 = 61;
int y1 = 37;
int x2 = 83;
int y2 = 43;
int x3 = 69;
int y3 = 60;
int x4 = 71;
int y4 = 82;
int x5 = 50;
int y5 = 73;
int x6 = 29;
int y6 = 82;
int x7 = 31;
int y7 = 60;
int x8 = 17;
int y8 = 43;
int x9 = 39;
int y9 = 37;
```

One array for each point

```
int[] p0 = { 50, 18 };
int[] p1 = { 61, 37 };
int[] p2 = { 83, 43 };
int[] p3 = { 69, 60 };
int[] p4 = { 71, 82 };
int[] p5 = { 50, 73 };
int[] p6 = { 29, 82 };
int[] p7 = { 31, 60 };
int[] p8 = { 17, 43 };
int[] p9 = { 39, 37 };
```

One array for each axis

```
int[] x = { 50, 61, 83, 69, 71, 50, 29, 31, 17, 39 };
int[] y = { 18, 37, 43, 60, 82, 73, 82, 60, 43, 37 };
```
Using Arrays to Draw

- The star has 10 vertex points, each with 2 values, for a total of 20 data elements.
- The code on the left demonstrates using 20 separate variables.
- The code in the middle uses 10 arrays, one for each point of the shape.
- The code on the right shows the data elements can be grouped logically together in two arrays, one for the x-coordinates and one for the y-coordinates.
Using Arrays to Draw

```java
int[] x = {50, 61, 83, 69, 71, 50, 29, 31, 17, 39};
int[] y = {18, 37, 43, 60, 82, 73, 82, 60, 43, 37};

smooth();
strokeWeight(3);
beginShape();
// Reads one array element every
// time through the for loop
for (int i = 0; i < x.length; i++) {
    vertex(x[i], y[i]);
}
endShape(CLOSE);
saveFrame();
```
int numLines = 12;
float[] x = new float[numLines];
float[] speed = new float[numLines];
float offset = 8;  // Set space between lines

void setup() {
  size(100, 100);
  smooth();
  strokeWeight(10);
  for (int i = 0; i < numLines; i++) {
    x[i] = i;  // Set initial position
    speed[i] = 0.1 + (i / offset);  // Set initial speed
  }
}
void draw() {
    background(204);
    for (int i = 0; i < x.length; i++) {
        x[i] += speed[i]; // Update line position
        if (x[i] > (width + offset)) { // If off the right,
            x[i] = -offset * 2; // return to the left
        }
        float y = i * offset; // Set y-coordinate
        line(x[i], y, x[i] + offset, y + offset); // Draw line
    }
}
Array Functions

- Processing provides a number of functions for manipulating arrays
- The `append()` function expands an array by one element, adds data to the new position, and returns the new array:

```java
String[] trees = { "ash", "oak" };
trees = append(trees, "maple"); // Add "maple" to the end
println("trees =");
println(trees); // Prints "ash oak maple"
```
Using the `append()` Function

```java
String[] trees = { "ash", "oak" };
append(trees, "maple"); // INCORRECT! Does not change the array
println("trees = ");
println(trees); // Prints "ash oak"

trees = append(trees, "maple"); // Add "maple" to the end
println("trees = ");
println(trees); // Prints "ash oak maple"

// Add "beech" to the end of the trees array, and creates a new // array to store the change
String[] moretrees = append(trees, "beech");
println("moretrees = ");
println(moretrees); // Prints "ash oak maple beech"
```
Array Functions

- The `shorten()` function decreases an array by one element by removing the last element and returns the shortened array.

```java
String[] trees = {"lychee", "coconut", "fig"};

trees = shorten(trees); // Remove last element from the array
println("trees = ");
println(trees); // Prints "lychee coconut"

trees = shorten(trees); // Remove last element from the array
println("trees = ");
println(trees); // Prints "lychee"
```
Array Functions

- The `expand()` function increases the size of an array.
  - It can expand to a specific size, or if no size is specified, the array’s length will be doubled.
  - If an array needs to have many additional elements, it’s faster to use `expand()` to double the size than to use `append()` to continually add one value.
Copying Arrays

‣ Array values cannot simply be copied with the assignment operator.
‣ One way to copy from one array to another is to write a `for` loop.
‣ The `arraycopy()` function is the most efficient way to copy from one array to another.
‣ The data is copied from the first array to the second array, given as parameters.
‣ Both arrays must be the same length.
String[] north = { "OH", "IN", "MI" };  
String[] south = { "GA", "FL", "NC" };  
arraycopy(north, south); // Copy from north array to south array  
println("south = ");  
println(south);       // Prints "OH IN MI"

String[] east = { "MA", "NY", "RI" };  
String[] west = new String[east.length]; // Create a new array  
arraycopy(east, west); // Copy from east array to west array  
println("west = ");  
println(west);       // Prints "MA NY RI"
Passing Arrays to Functions

- New functions can be written to perform operations on arrays, but arrays behave differently than data types such as int and char.
- When an array is used as a parameter to a function, the address (location in memory) of the array is transferred into the function instead of the actual data. No new array is created, and changes made within the function affect the array used as the parameter.
float[] data = {19.0, 40.0, 75.0, 76.0, 90.0};

void setup() {
    halve(data);
    println(data[0]); // Prints "9.5"
    println(data[1]); // Prints "20.0"
    println(data[2]); // Prints "37.5"
    println(data[3]); // Prints "38.0"
    println(data[4]); // Prints "45.0"
}

void halve(float[] d) {
    for (int i = 0; i < d.length; i++) { // For each array element,
        d[i] = d[i] / 2.0; // divide the value by 2
    }
}
float[] data = {19.0, 40.0, 75.0, 76.0, 90.0};
float[] halfData;

void setup() {
    halfData = halve(data);       // Run the halve() function
    println(data[0] +", " + halfData[0]); // Prints "19.0, 9.5"
    println(data[1] +", " + halfData[1]); // Prints "40.0, 20.0"
    println(data[2] +", " + halfData[2]); // Prints "75.0, 37.5"
    println(data[3] +", " + halfData[3]); // Prints "76.0, 38.0"
    println(data[4] +", " + halfData[4]); // Prints "90.0, 45.0"
}

float[] halve(float[] d) {
    float[] numbers = new float[d.length];     // Create a new array
    arraycopy(d, numbers);                     // Copy the contents
    for (int i = 0; i < numbers.length; i++) {  // For each element,
        numbers[i] = numbers[i] / 2;            // divide the value by 2
    }
    return numbers;                            // Return the new array
}
Two-Dimensional Arrays

- Data can also be stored and retrieved from arrays with more than one dimension.
- A 2D array is essentially an array of arrays. It must be declared, then created, and then the values can be assigned just as in a 1D array.
- The data points for the star can be put into a 2D array instead of two 1D arrays:
int[][] points = {{50, 18}, {61, 37}, {83, 43}, {69, 60}, {71, 82}, {50, 73}, {29, 82}, {31, 60}, {17, 43}, {39, 37}};

void setup() {
  size(100, 100);
  fill(0);
  smooth();
}

void draw() {
  background(204);
  translate(mouseX - 50, mouseY - 50);
  beginShape();
  for (int i = 0; i < points.length; i++) {
    vertex(points[i][0], points[i][1]);
  }
  endShape();
}
Using Arrays for Animation

- Arrays can be used to store and draw a sequence of images to produce an animation.
  - The frames of the animation should be loaded in the `setup()` function and the `draw()` function should choose one image to draw on each frame.
int numFrames = 12;  // The number of animation frames
PImage[] images = new PImage[numFrames];  // Image array

void setup() {
    size(100, 100);
    frameRate(30);  // Maximum 30 frames per second

    // Automate the image loading procedure. Numbers less than 100
    // need an extra zero added to fit the names of the files.
    for (int i = 0; i < images.length; i++) {
        // Construct the name of the image to load
        String imageName = "ani-" + nf(i, 3) + ".gif";
        images[i] = loadImage(imageName);
    }
}

void draw() {
    // Calculate the frame to display, use % to cycle through frames
    int frame = frameCount % numFrames;
    image(images[frame], 0, 0);
}
int numFrames = 12; // The number of animation frames
int topFrame = 0; // The top frame to display
int bottomFrame = 0; // The bottom frame to display
PImage[] images = new PImage[numFrames];
int lastTime = 0;

void setup() {
    size(100, 100);
    frameRate(30);
    for (int i = 0; i < images.length; i++) {
        images[i] = loadImage("ani-" + nf(i, 3) + ".gif");
    }
}

void draw() {
    topFrame = (topFrame + 1) % numFrames;
    image(images[topFrame], 0, 0);
    if ((millis() - lastTime) > 500) {
        bottomFrame = (bottomFrame + 1) % numFrames;
        lastTime = millis();
    }
    image(images[bottomFrame], 0, 50);
}
Pixels
Getting Pixel Values

- The `get()` function can read the color of any pixel in the display window.
- It can also grab the whole display window or a section of it. There are three versions of this function, one for each use.

```plaintext
// Returns a copy of display window as a PImage
get()
// Returns color value of a single pixel
get(x, y)
// Returns a rectangular area as a PImage
get(x, y, width, height)
```
Getting the Display Window

strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage cross = get(); // Get the entire window
image(cross, 0, 50); // Draw the image in a new position
Getting a Pixel Rectangle

strokeWeight(8);
line(0, 0, width, height);
line(0, height, width, 0);
PImage slice = get(0, 0, 20, 100); // Get window section
set(18, 0, slice);
set(50, 0, slice);
Getting Single Pixel Values

PImage trees;
trees = loadImage("topanga.jpg");
nostroke();
image(trees, 0, 0);
color c = get(20, 30);  // Get color at (20, 30)
fill(c);
rect(20, 30, 40, 40);
PImage trees;

int y = 0;

void setup() {
    size(100, 100);
    trees = loadImage("topangaCrop.jpg");
}

void draw() {
    image(trees, 0, 0);
    y = frameCount % height;
    for (int i = 0; i < 49; i++) {
        color c = get(i, y);
        stroke(c);
        line(i + 50, 0, i + 50, 100);
    }
    stroke(255);
    line(0, y, 49, y);
}
Getting Pixels From an Image

- Every PImage variable has its own `get()` to grab pixels from the image.
- This allows pixels to be grabbed from an image independently of the pixels in the display window.
- Because a PImage is an object, the `get()` function is accessed with the name of the image and the dot operator.
Getting Pixels From an Image

```java
PImage trees;
trees = loadImage("topanga.jpg");
stroke(255);
strokeWeight(12);
image(trees, 0, 0);
line(0, 0, width, height);
line(0, height, width, 0);
PImage treesCrop = trees.get(20, 20, 60, 60);
image(treesCrop, 20, 20);
```
Writing Pixel Values

- The pixels in Processing’s display window can be written directly with the `set()` function.
- There are two versions of this function, each with three parameters.

```cpp
// Set the color of a single pixel
set(x, y, color)

// Set the pixel colors of a rectangle using a PImage
set(x, y, image)
```
Writing Single Pixels

```java
for (int i = 0; i < 55; i++) {
    for (int j = 0; j < 55; j++) {
        color c = color((i + j) * 1.8);
        set(30 + i, 20 + j, c);
    }
}
```
Using `set()` to Draw Images

- The `set()` function can write an image to the display window at any location.
- Using `set()` to draw an image is faster than using the `image()` function because the pixels are copied directly.
- However, images drawn with `set()` cannot be resized or tinted, and they are not affected by the transformation functions.
Using `set()` to Draw Images

```java
PImage trees;

void setup() {
    size(100, 100);
    trees = loadImage("topangaCrop.jpg");
}

void draw() {
    int x = frameCount % 50;
    set(x, 0, trees);
}
```
Writing Pixels in Images

- Every PImage variable has its own set() function to write pixels directly to the image.
  - This allows pixels to be written to an image independently of the pixels in the display window.
  - Because a PImage is an object, the set() function is run with the name of the image and the dot operator.
Writing Pixels to Images

PImage trees;

void setup() {
    trees = loadImage("topangaCrop.jpg");
}

void draw() {
    background(0);
    int x = frameCount % trees.width;
    int y = frameCount % trees.height;
    color white = color(255);
    trees.set(x, y, white);
    image(trees, 20, 0);
}
Motion
Easing

• Easing, also called interpolation, is a technique for moving between two points.
  • By moving a fraction of the total distance each frame, a shape can decelerate (or accelerate) as it approaches a target location.
float x = 20.0; // Initial x-coordinate
float y = 10.0; // Initial y-coordinate
float targetX = 70.0; // Destination x-coordinate
float targetY = 80.0; // Destination y-coordinate
float easing = 0.05; // Size of each step along the path

void setup() {
  size(100, 100);
  noStroke();
  smooth();
}

void draw() {
  fill(0, 12);
  rect(0, 0, width, height);
  float d = dist(x, y, targetX, targetY);
  if (d > 1.0) {
    x += (targetX - x) * easing;
    y += (targetY - y) * easing;
  }
  fill(255);
  ellipse(x, y, 20, 20);
}
The simple curves covered in Lecture 4 can provide paths for shapes in motion.

Instead of drawing the entire curve in one frame, it’s possible to calculate each step of the curve on consecutive frames.

The following example presents a very general way to write this code. Changing the variables at the top of the code changes the start and stop position, the curve shape, and the number of steps to take along the curve.
float beginX = 20.0;       // Initial x-coordinate
float beginY = 10.0;       // Initial y-coordinate
float endX = 70.0;         // Final x-coordinate
float endY = 80.0;         // Final y-coordinate
float distX;               // X-axis distance to move
float distY;               // Y-axis distance to move
float exponent = 0.5;      // Determines the curve
float x = 0.0;             // Current x-coordinate
float y = 0.0;             // Current y-coordinate
float step = 0.01;         // Size of each step along the path
float pct = 0.0;           // Percentage traveled (0.0 to 1.0)
void setup() {
    size(100, 100);
    noStroke();
    smooth();
    distX = endX - beginX;
    distY = endY - beginY;
}

void draw() {
    fill(0, 2);
    rect(0, 0, width, height);
    pct += step;
    if (pct < 1.0) {
        x = beginX + (pct * distX);
        y = beginY + (pow(pct, exponent) * distY);
    }
    fill(255);
    ellipse(x, y, 20, 20);
}
int direction = 1;

void draw() {
    fill(0, 2);
    rect(0, 0, width, height);
    pct += step * direction;
    if ((pct > 1.0) || (pct < 0.0)) {
        direction = direction * -1;
    }
    if (direction == 1) {
        x = beginX + (pct * distX);
        float e = pow(pct, exponent);
        y = beginY + (e * distY);
    } else {
        x = beginX + (pct * distX);
        float e = pow(1.0 - pct, exponent * 2);
        y = beginY + (e * -distY) + distY;
    }
    fill(255);
    ellipse(x, y, 20, 20);
}
As a shape moves along a curve, its speed changes.

A curve can be used to control the speed of a visual element that moves on a straight line.
void draw() {
    fill(0, 2);
    rect(0, 0, width, height);
    if (pct < 1.0) {
        pct = pct + step;
        float rate = pow(pct, exponent);
        x = beginX + (rate * distX);
        y = beginY + (rate * distY);
    }
    fill(255);
    ellipse(x, y, 20, 20);
}
Motion Through Transformation

- The transformation functions can also create motion by changing the parameters to `translate()`, `rotate()`, and `scale()`.
- Transformations reset at the beginning of each `draw()`. Calling `translate(5, 0)` will always move the coordinate system 5 pixels to the right in each frame. It will not move the system 5 right on the first frame, 10 on the next, 15 on the next etc.
float y = 50.0;
float speed = 1.0;
float radius = 15.0;

void setup() {
    size(100, 100);
    smooth();
    noStroke();
    ellipseMode(RADIUS);
}

void draw() {
    fill(0, 12);
    rect(0, 0, width, height);
    fill(255);
    translate(0, y); // Set the y-coordinate of the circle
    ellipse(33, 0, radius, radius);
    y += speed;
    if (y > height + radius) {
        y = -radius;
    }
}
float angle = 0.0;

void setup() {
    size(100, 100);
    smooth();
    noStroke();
}

void draw() {
    fill(0, 12);
    rect(0, 0, width, height);
    fill(255);
    angle = angle + 0.02;
    translate(70, 40);
    rotate(angle);
    rect(-30, -30, 60, 60);
}
The \textit{sin()} function is often used to produce elegant motion. It can generate an accelerating and decelerating speed as a shape moves from one frame to another.
float angle = 0.0; // Current angle
float speed = 0.1; // Speed of motion
float radius = 40.0; // Range of motion

void setup() {
    size(100, 100);
    noStroke();
    smooth();
}

void draw() {
    fill(0, 12);
    rect(0, 0, width, height);
    fill(255);
    angle += speed;
    float sinval = sin(angle);
    float yoffset = sinval * radius;
    ellipse(50, 50 + yoffset, 80, 80);
}
Periodic Motion

- Adding values from \( \sin() \) and \( \cos() \) can produce more complex movement that remains periodic.

- In the following example, a small dot moves in a circular pattern using values from \( \sin() \) and \( \cos() \). A larger dot uses the same values for its base position but adds additional \( \sin() \) and \( \cos() \) calculations to produce an offset.
float angle = 0.0; // Current angle
float speed = 0.05; // Speed of motion
float radius = 30.0; // Range of motion
float sx = 2.0;
float sy = 2.0;

void setup() {
    size(100, 100);
    noStroke();
    smooth();
}
void draw() {
    fill(0, 4);
    rect(0, 0, width, height);
    angle += speed; // Update the angle
    float sinval = sin(angle);
    float cosval = cos(angle);

    // Set the position of the small circle based on new
    // values from sine and cosine
    float x = 50 + (cosval * radius);
    float y = 50 + (sinval * radius);
    fill(255);
    ellipse(x, y, 2, 2); // Draw smaller circle

    // Set the position of the large circles based on the
    // new position of the small circle
    float x2 = x + cos(angle * sx) * radius / 2;
    float y2 = y + sin(angle * sy) * radius / 2;
    ellipse(x2, y2, 6, 6); // Draw larger circle
}
Phase Shifting

- The phase of a function is one iteration through its possible values—for example, a single rise and fall sequence of a sine curve.

- Phase shifting occurs when the function is offset to start at a different point within the phase, such as the downward part of a sine curve rather than the top.
float angle = 0.0;
float speed = 0.1;

void setup() {
    size(100, 100);
    noStroke();
    smooth();
}

void draw() {
    background(0);
    angle = angle + speed;
    ellipse(50 + (sin(angle + PI) * 5), 25, 30, 30);
    ellipse(50 + (sin(angle + HALF_PI) * 5), 55, 30, 30);
    ellipse(50 + (sin(angle + QUARTER_PI) * 5), 85, 30, 30);
}
Organic Motion

- Examples of organic movement include a leaf falling, an insect walking, a bird flying, a person breathing, a river flowing, and smoke rising.
- This type of motion is often considered idiosyncratic and stochastic.
- It is often non-periodic or “almost” periodic.
float x = 50.0; // X-coordinate
float y = 80.0; // Y-coordinate

void setup() {
    size(100, 100);
    randomSeed(0); // Force the same random values
    background(0);
    stroke(255);
}

void draw() {
    x += random(-2, 2); // Assign new x-coordinate
    y += random(-2, 2); // Assign new y-coordinate
    point(x, y);
}
Position and Direction

- The \texttt{sin()} and \texttt{cos()} functions can be used to create unpredictable motion when employed with the \texttt{random()} function.

- The following example the position of the line at each frame is based on its current position and the slight variation to its direction. The \texttt{cos()} function uses the angle to calculate the next \texttt{x-coordinate} for the line, and the \texttt{sin()} function uses the same angle to calculate the next \texttt{y-coordinate}. 
float x = 0.0; // X-coordinate
float y = 50.0; // Y-coordinate
float angle = 0.0; // Direction of motion
float speed = 0.5; // Speed of motion

void setup() {
    size(100, 100);
    background(0);
    stroke(255, 130);
    randomSeed(121); // Force the same random values
}

void draw() {
    angle += random(-0.3, 0.3);
    x += cos(angle) * speed; // Update x-coordinate
    y += sin(angle) * speed; // Update y-coordinate

    translate(x, y);
    rotate(angle);
    line(0, -10, 0, 10);
}
Animating Transformations

‣ In the following example, the angle variable for the \texttt{tail()} function is continually changing to produce a swaying motion.

‣ Because the angles for each shape accumulate with each unit, the longest shapes with the most units swing from side to side with a greater curvature.
void draw() {
    background(0);
    inc += 0.01;
    float angle = sin(inc) / 10.0 + sin(inc * 1.2) / 20.0;
    tail(18, 9, angle / 1.3);
    tail(33, 12, angle);
    tail(44, 10, angle / 1.3);
    tail(62, 5, angle);
    tail(88, 7, angle*2);
}

void tail(int x, int units, float angle) {
    pushMatrix();
    translate(x, 100);
    for (int i = units; i > 0; i--) {
        strokeWeight(i);
        line(0, 0, 0, -8);
        translate(0, -8);
        rotate(angle);
    }
    popMatrix();
}
Lab Exercises
Lab Exercises

- Create an array to store the y-coordinates of a sequence of shapes.
  - Draw each shape inside a for loop and use the values from the array to set the y-coordinate.

- Write a function to multiply two arrays together & return the result as a new array.
  - Print the results to the console.
Lab Exercises

- Use a 2D array to store the coordinates for a shape of your own invention.
  - Use a for structure to draw the shape to the display window.

- Load a sequence of related images into an array and use them to create an animation.
  - Modify the program to present each frame of animation in a random sequence.
Lab Exercises

‣ Move two shapes continuously, but constrain their positions to the display window.

‣ Move three shapes on different curves to create a kinetic composition.

‣ Use the transformation functions to animate a shape.
Lab Exercises

- Make a shape move with numbers returned from \( \sin() \) and \( \cos() \).
- Develop a kinetic composition using the concept of phase shifting.
- Use code 32-06 as a base for creating a more advanced organism.
Lab Exercises

・Load an image and use `get()` to create a collage by overlaying different sections of the same image.

・Draw a shape in the display window. Copy a section of the window to another by using `get()` and `set()` within a `for` loop.