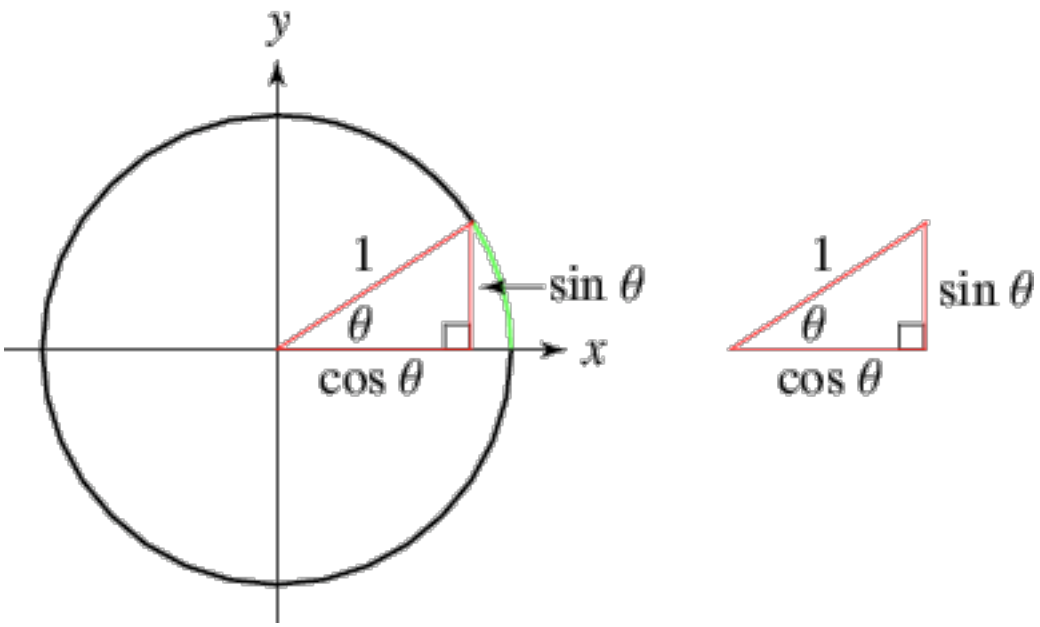
Nature of Code

Patrick Dwyer
Fall 2005
Week 4 - September 27th

Trigonometry

Trigonometry is the study and utilization of the relationships between the sides and angles of triangles. We'll be using sines and cosines, as well as the arctangent. For a review of basic trigonometry visit the Trigonometry section of Mathworld (http://mathworld.wolfram.com/Trigonometry.html).



Cartesian and Polar Coordinates

So far in our programs we've been modeling motion and simulation in a Cartesian Coordinate plane; each object has an X and Y value that correspond to the horizontal and vertical on the screen. We can also take advantage of Polar Coordinate planes. Polar coordinates are a pair of values like X and Y, but instead of describing a horizontal and vertical value, they describe a rotation and a radius.

The rotation is an angle described in Radians, while the radius is a normal floating point number. Using basic trigonometry we can convert back and forth between Cartesian and Polar coordinates:

```
float r = 50.0f;
float theta = radians(30);

// convert from polar to cartesian
float x = r * cos(theta);
float y = r * sin(theta);
println("X: " + x + ", Y: " + y);
```

```
// cartesian to polar
r = sqrt( x * x + y * y);
theta = arctan2(y, x);
println("Theta: " + theta + ", R: " + r);
```

Processing only knows how to work with Cartesian coordinates (X and Y values, like pixel position), so when we do work in Polar coordinates we'll need to convert them to Cartesian coordinates to draw them on screen. The benefit of using polar coordinates is in creating objects that can have angular velocity and acceleration, among other things.

Oscillation

An oscillation is the periodic movement between two points, as when a pendulum swings back and forth. In our programs we can simulate oscillations using harmonic motion. The simple harmonic motion we'll be using (the periodic oscillations of an object based on sines or cosines) can be based upon an X location changing as a function of time. To construct our harmonic motion we'll use three components:

- **Amplitude**: The distance from the center point of the motion to either extreme.
- **Period**: The amount of time or distance it takes to complete one cycle of motion.
- **Frequency**: The number of cycles over time (usually expressed as 1/period).

The X position as a function of time can be determined with the equation:

```
x(t) = Amplitude * cosine( 2 * PI * t / Period)
```

In our programs we can consider time to be counted as frames of animation, with `t` incrementing by one each frame. If we assume we have previously defined a variable called framecounter that increments with each frame, we can oscillate X between -75 and 75 using the following code:

```
float period = 600.0f;
float amplitude = 75.0f;
float x = amplitude * cos(TWO_PI * framecounter / period);
framecounter++;
```

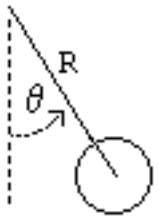
We could also use angular velocity to easily describe this type of oscillation. Assume that we have already defined `theta` and `theta_vel`:

```
float amplitude = 75.0f;
float x = amplitude * sin(theta);
theta += theta_vel;
```

Here the period and frequency of the oscillation are tied directly to angular velocity. So an object with an angular velocity of 10 with have twice the period (twice as long to complete a full oscillation) as an object with an angular velocity of 20. As well, it would have half the frequency.

Angular velocity is possibly a much easier way to simulate oscillating movement, and is reminiscent of the Vector forces we covered in the first two weeks.

Pendulums



We described oscillating motion as being similar to a pendulum swinging, and the examples we've seen so far can help us simulate a pendulum. For a pendulum we have a gravitational force pulling down, but the pendulum can only move as allowed by it's arm. To simulate the pendulum we consider the pendulum's angle and evaluate the changes in motion based upon angular acceleration and velocity.

Given a pendulum with arm angle theta (0 being the pendulum at rest) and an arm radius of r, we can use sine to calculate the angular force exerted by gravity upon the pendulum.

$F_{gravity}$ = Mass * Gravity
$F_{pendulum}$ = Fgravity * sine(theta)
Angular Acceleration = $F_{pendulum}$ / Mass = Gravity sine(theta)

Like our other simulations, this example is based upon an idealized (faked) world, where there is no tension or resistance in the pendulum. More on pendular movement can be found at:
http://www.myphysicslab.com/pendulum1.html

Using our formulate for pendular motion, we can write a Class for Processing that encapsulates the necessary properties for a pendulum:

```
class Pendulum {

    // Our pendulum uses a vector to calculate the location
    // of the ball at the end of the pendulum arm
    Vector2D loc;

    // Another vector keeps track of the fulcrum of the pendulum;
    // the fixed point from which it swings.
    Vector2D fulcrum;

    // We'll be working in polar coordinates for our
    // pendulum motion, so keep track of our radius
    // and our movement of theta
    float r;
    float theta;
    float theta_vel;
    float theta_accel;
... ect
```

You can refer to the Pendulum example for the full code of the class. To translate our sample equation for calculating the motion of the pendulum to Processing, we'll use an `update` method in our Pendulum class:

```
    void update() {
            // use an arbitrary value for gravity
            float G = 0.4;
```

```
        // calculate the angular acceleration due to gravity
        theta_accel = (-1 * G / r) * sin(theta);

        // apply the acceleration to our movement
        theta_vel += theta_accel;
        theta_vel *= dampen;
        theta += theta_vel;

        // recalculate our position
        loc.setX(r * sin(theta));
        loc.setY(r * cos(theta));
        loc = loc.add(fulcrum);
    }
```

Waves

To start working with waves, we'll graph a sine function. We need to consider the following properties of the waves we want to simulate:

- **Amplitude**: The height of the wave (measure on the Y axis)
- **X spacing**: Number of pixels between each point on the x axis
- **Width**: Total pixel width of the entire wave
- **Period**: Number of pixels to cover one cycle of the wave

To simulate our wave we'll be using an array of height values; one height value for each point along the x-axis, calculated using a sine or cosine. A sine or cosine wave has a Period of 360 degrees (two PI), so we calculate how our x values will increment according to this Period:

```
dx * ( period / spacing) = 2 * PI
dx = (2 * PI * spacing) / period
```

Using this value we can calculate the height of our wave across our given number of samples (x values):

```
int xspacing = 8;
int w = width;
float period = 100.0f;
float amplitude = 50.0f;
float dx = (TWO_PI / period) * xspacing;
float[] yvalues = new float[w / xspacing];

// use a for loop to go through all of our xvalues
// across the screen and calculate the height of the wave
float x = 0.0f;
for (int i = 0; i < yvalues.length; i++) {
    yvalues[i] = sin(x) * amplitude;
    x += dx;
```

```
}
```

Once we've calculated and stored the height values for our wave, we can graph them on the screen however we choose. We could have drawn the graph to the screen instead of storing the values, but it is more flexible and cleaner to separate the calculate and drawing. Here's one way we could graph the sine wave, albeit a boring one:

```
for (int x = 0; x < yvalues.length; x++) {
      noStroke();
      fill(0, 0, 255, 50);
      ellipseMode(CENTER);
      ellipse(x * xspacing, yvalues[x], 15, 15);
}
```

This example, using the two code snippits, will draw a "static" wave, one that doesn't move. To add motion to the wave, we can simply chnage the x value used to calculate the height of the wave to a different "angle". Take a look at the example for graphing a basic wave for a simple version of wave motion.

Complex Waves

We've drawn very simple, very predictable waves using sine and cosine. To make more interesting, and far less predictable waves we can use two approaches:

- **Perlin Noise:** We can take advantage of the techniques we developed for using Perlin noise to create interesting wave patterns that look almost random:

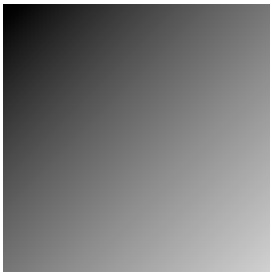
```
for (int i = 0; i < yvalues.length; i++) {
      // you could add a second and third dimension to the noise
      // to make it even more interesting
      float n = 2 * noise(x) - 1.0f;
      yvalues[i] = n * amplitude;
      x += dx;
}
```

- **Additive Waves:** By combining two or more waves together, each with different amplitudes and frequencies, we can create much more complex wave patterns. The Add Waves example includes code for creating a collection of waves that are combined for a single complex wave pattern.

2D Graphing

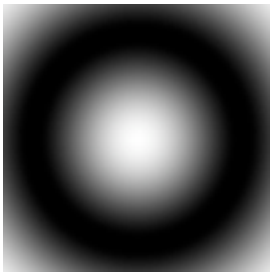
When we first worked with Perlin noise we discovered that we can create interesting visualizations and patterns by visiting every pixel on the screen and calculating a value for it, based on its X and Y position. We can create similar effects using trigonometry. Take the following example:

```
size(100, 100);
loadPixels();
for (int x = 0; x < width; x++) {
      for (int y = 0; y < height; y++) {
            pixels[x + y * width] = color(x + y);
      }
}
updatePixels();
```



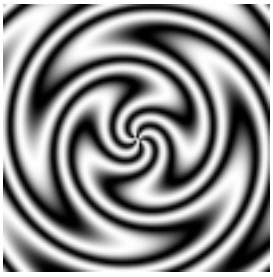
Here we've created a grayscale color for every X and Y position, resulting in a simple gradient effect. If we modify this to graph trigonometry functions based on polar coordinates, we can make things much more interesting. First, we need to map the screen to coordinates that are useful in our functions. We want the top left corner of the screen to represent (-4, -4) and the bottom right to represent (4, 4) for our functions.

```
// width and height of our 2D space
float w = 8.0f;
float h = 8.0f;

// how much to X and Y change per pixel
float dx = w / width;
float dy = h / height;

loadPixels();
// our 2D x space starts at -4
float x = -w/2;
for (int i = 0; i < width; i++) {

      // our 2D y space starts at -4
      float y = -h/2;       //start y at -1 * height / 2
      for (int j = 0; j < height; j++) {

            // convert our coordinates into polar coordinates
            float r = sqrt((x*x) + (y*y));
            float theta = atan2(y,x);

            // create our value at this pixel
            float val = cos(r);

            // scale the value to a color values
            pixels[i+j*width] = color((val + 1.0f) * (255.0/2));
            y += dy;
      }
      x += dx;
}
updatePixels();
```



A more interesting value can be generated using this formula for the val variable:

```
float val = sin(6 * cos(r) + 5 * theta);
```



Assignment

Develop an idea for a Two Week assignment. The assignment should be divided into two parts. For next week post your progress with the first part of the assignment, as well as a written description of your goal. The second part will be presented in class in two weeks.

Ideas:

- Take the techniques we've learned so far (velocity, acceleration, randomness, noise, oscillation) and create a "creature" that moves, guided by various rules. For the second part of the assignment extend the creature into a particle system of multiple creatures.
- Take one of the simulations we've created so far, and leaving the algorithms that define the simulation intact, develop a new way of visualizing the system; change only the drawing and rendering code.
- Combine linear and oscillating movement. Create a body that moves through a space guided by linear (velocity, acceleration) motion, and use that body as the center point for an oscillating movement of another body.