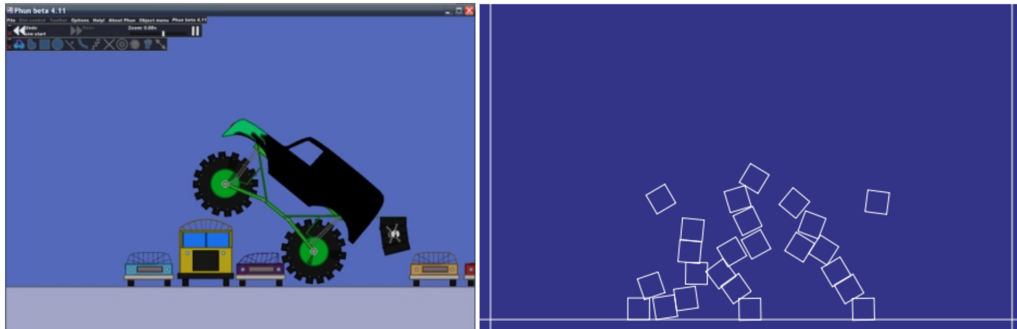
Chapter 3

Simulation and User Interaction

What is in This Chapter ?

This chapter explains the concepts behind very simple **computer simulations**. All examples are explained within a graphical context. The idea of a processing loop is explained and how various simple 1D and 2D motion-related simulations can be programmed. Next, user interaction is explained within the context of **event handling** within an **event-loop**. Examples are given showing how to handle **mouse-related events**.



3.1 Simulation

Computers are often used to simulate real-world scenarios. Here is the wikipedia definition:

> *A **computer simulation** is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behaviour of the system.*

Computer simulation has become a useful part of modeling many natural systems in the areas of physics, chemistry, biology, economics, social science and engineering.

Video games are prime examples of simulation, where some real (or imaginary) life situations are simulated in a virtual world. As time progresses, video games are becoming more life-like as graphics and physical modeling become more precise and realistic.

Other places that require computer simulation are:

- network traffic analysis
- city/urban simulation
- flight/space/vehicle/medical simulators
- disaster preparedness simulations
- film and movie production
- theme park rides
- manufacturing systems

There is much to know about simulation, enough material to fill a course or two. In this course, however, we will just address two basic categories of simulations:

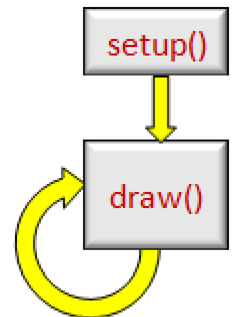
- Running a simulation to find an answer to a problem
- Virtual simulation (i.e., animation) of a real world scenario

When simulating, usually there is some kind of ***initialization*** phase, followed by a ***processing loop*** that continuously processes and (possibly) displays something on the screen.

Think of a simulation as a store. The ***initialization*** phase corresponds to the work involved in getting the store ready to open in the morning (e.g., tidying up, putting out signs, stocking the shelves, preparing the cash register, unlocking the door). The ***processing loop*** phase corresponds to the repeated events that occur during the day (i.e., dealing with customers one-by-one) until the store is to be closed.

In Processing, the **setup()** procedure is called once at the beginning of your program and it represents the initialization phase of your program. It is also used as the main entry point of your program (i.e., your program starts there).

The processing loop phase is accomplished via the **draw()** procedure … which is called repeatedly when your program is running. It is useful for performing repeated computations and displaying information as well as interacting with the user (e.g., via the mouse). By default, the **draw()** procedure is called directly after the **setup()** procedure has been evaluated. Your program does not need to have a **draw()** procedure, but if it is there, your program will call it.



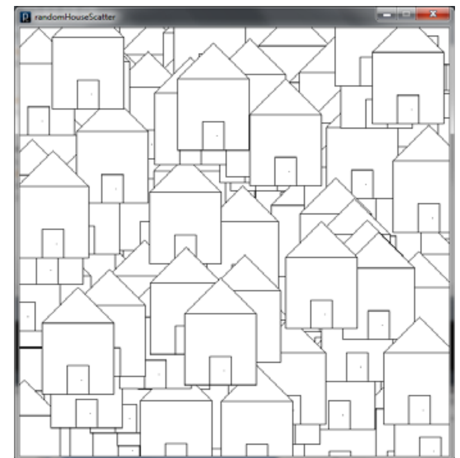
Example:

Recall the **drawHouse()** method that we wrote earlier. Here is how we could repeatedly draw houses at random locations on the screen:

```
void setup() {
     size(600,600);
}

void draw() {
     drawHouse(int(random(width))-50,
               int(random(height))+50);
}

void drawHouse(int x, int y) {
     rect(x, y-100,100,100);
     triangle(x,y-100,(x+50),y-150,(x+100),y-100);
     rect(x+35,y-40,30,40);
     point(x+55,y-20);
}
```

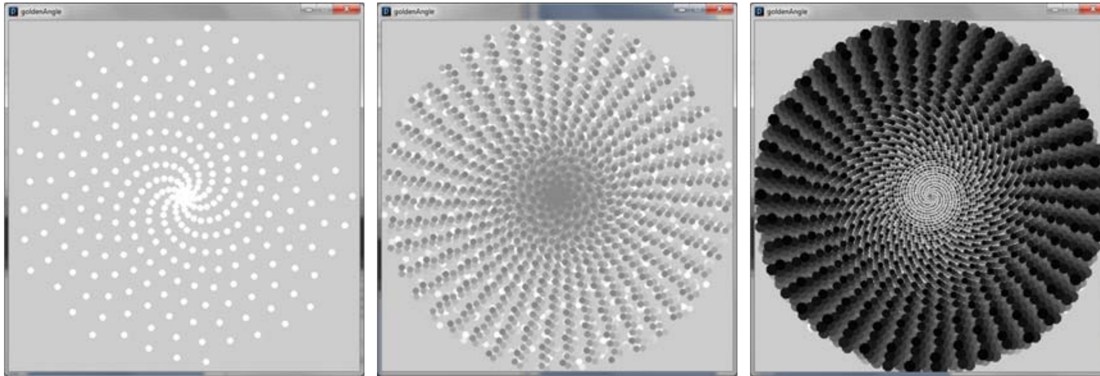


The program represents the simplest kind of simulation … that of simply displaying something in an endless loop. Notice that the **draw()** procedure simply calls **drawHouse()** with a random **x** value from 0 to width=600 and a random **y** value from 0 to height=600 as well. Remember, the **draw()** method is called repeatedly, so the program will continually (and endlessly) draw the house at random positions.

Of course, we could vary the above program to make random sized houses with random colors or we could have it draw the houses along a path (e.g., spiral). Regardless of what we are drawing and how we are drawing it, the programs will all have the same notion of a repeated **draw()** loop.

Example:

Here is a more computational example that displays a spiraling sequence of circles.



You may notice a few differences in the code from the previous example:

```
int      radius;         // distance from center to draw from
float    angle;          // angle to draw at
int      grayLevel;      // color to display each round of spirals

void setup() {
   size(600,600);
   radius = 0;
   angle = 0;
   grayLevel = 0;
}

void draw() {
   stroke(255-grayLevel, 255-grayLevel, 255-grayLevel);
   fill(255-grayLevel, 255-grayLevel, 255-grayLevel);
   ellipse(int(cos(angle)*radius)+width/2,
           int(sin(angle)*radius) + height/2,
           10, 10);

   angle = angle + 137.51;
   radius = (radius+1) % 300;
   if (radius == 0)
       grayLevel = (grayLevel + 10) % 255;
}
```

There are three variables declared at the top of the program. These are declared outside the **setup()** and **draw()** procedures because they are used in both procedures. Hence, they are global variables. The **radius** represents the distance (from the center) that we are drawing the circles at, while the **angle** represents the angle that we are drawing them at. The **grayLevel** indicates the color that the spirals are drawn at, which begins with white and gets darker for each round of spirals.

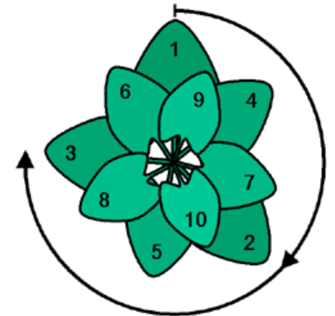
You may notice that the **draw()** procedure first draws the circle (with appropriate gray-level fill and border) and then simply updates the angle and radius for the next time that the **draw()** procedure is called. The **%** operator is the ***modulus operator*** that gives the remained after dividing by a specified number. The modulus operator is great for ensuring that an integer does not exceed a certain value but that it begins again at 0. The following two pieces of code do the same thing:

`radius = (radius + 1) % 255;`	`radius = radius + 1;`
	`if (radius >= 255)`
	`    radius = 0;`

For example if **x** was initially 0 and then we did **x = (x + 1)%5**, here would be the values for **x**:

0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2 … etc..

The angle of **137.51** is called the ***golden angle*** as it is found in nature as the ideal angle for producing spirals as shown in the design of seashells, flower petals, etc.. The angle is ideal as it minimizes overlap during multiple rounds of spiraling.



While this example illustrates the ability to vary the computational parameters during the processing loop of the simulation, it really does not serve any particular purpose other than to produce a nice picture. Now let us look at an example that actually attempts to compute something interesting:

Example:

It is often the case that we need to compute an answer to some problem in which the parameters are complex and/or uncertain. In some situations, it may be unfeasible or impossible to compute an exact result to a problem using a well-defined and predictable algorithm. There is a specific type of simulation method that is well-suited for situations in which you need to make an estimate, forecast or decision where there is significant uncertainty:

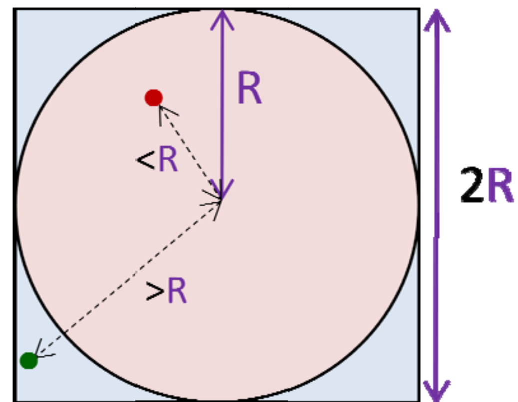
> *The **Monte Carlo Method** uses randomly generated or sampled data and computer simulations to obtain approximate solutions to complex mathematical and statistical problems.*

There is always some error involved with this scheme, but the larger the number of random samples taken, the more accurate the result.

The simplest example that is used to describe the Monte Carlo method is that of computing an approximation of **π** (i.e., **pi**). **π** is a mathematical constant whose value is the ratio of any circle's circumference to its diameter. It is approximately equal to **3.141593** in the usual decimal notation. **π** is a very important number in math and computer science as it relates to many trigonometric functions and geometric algorithms and is used in graphics and animation.

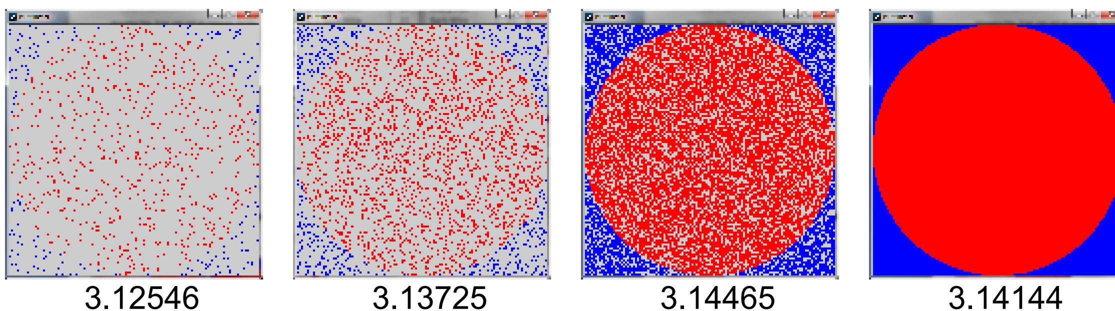
The value of π can be approximated using a Monte Carlo method based on this principle:

> **Given a circle inscribed in a square (i.e., the largest circle that fits in the square), the ratio of the area of the circle to that of the square is π / 4.**



Knowing this, if we can get an estimate for the area of the circle as well as the area of the square, then we can find an approximation for **π / 4**, of course then multiplying by **4** to get an approximation for **π**.

We can estimate the area of the square and circle by uniformly scattering some points throughout the square. Some will lie within the circle, some will lie outside the circle. The more points that we add, the better the approximation of the area, as the whole square and circle will eventually be covered as time goes on.



Here is the algorithm, assuming that the circle and square are centered at point **(R,R)**:

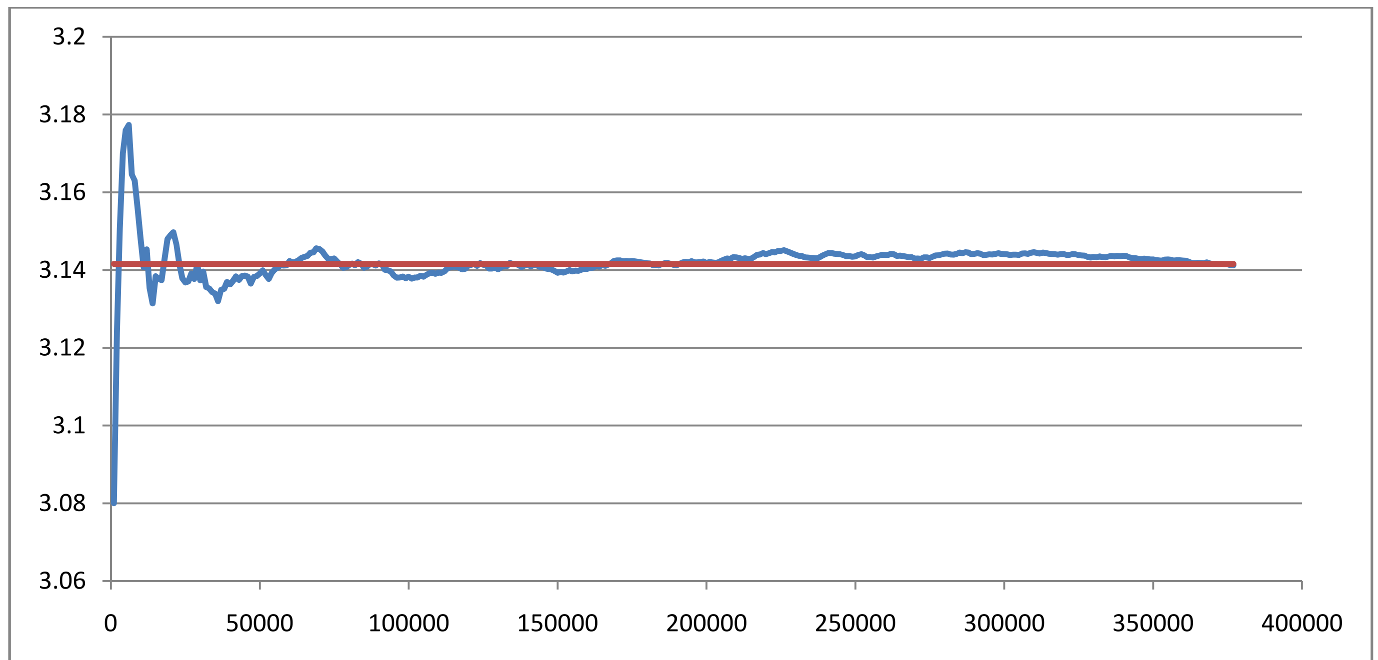
```
Algorithm: ComputePi
    R:            the radius of a circle and ½ width & ½ height of a square

1.  pointsInCircle ← 0
2.  pointsInSquare ← 0
3.  repeat for a user-chosen amount of iterations {
4.      x ← random value from 0 to 2R-1
5.      y ← random value from 0 to 2R-1
6.      pointsInSquare ← pointsInSquare + 1
7.      if ((the distance from (x,y) to (R,R) < R) then
8.              pointsInCircle ← pointsInCircle + 1
    }
9.  print (pointsInCircle / pointsInSquare * 4)
```

We can stop the loop at any time. As the loop goes on, however, the algorithm will slowly convergence to a better approximation as more data points are sampled. If the points are purposefully chosen only around the center of the circle, they will not be uniformly distributed, and so the approximation will be poor. An approximation will also be poor if only a few points

are randomly chosen throughout the whole square. Thus, the approximation of π will become more accurate both with more points and with their uniform distribution.

Here is an example showing how the approximated value (blue) will converge towards the optimal value (solid red line).



In processing, we can write a program that shows us visually what is happening by drawing each point that is randomly chosen. For those inside the circle we can draw them as red and those outside as blue. We can use the **draw()** procedure as our **repeat** loop from our above algorithm since it repeats "forever" (or until the program is manually stopped).
Here is the corresponding Processing code:

```
int     pointsInCircle = 0;
int     pointsInSquare = 0;
int     R = 250;

void setup() {
  size(2*R,2*R);
}

void draw() {
  int x, y;
  x = (int)random(2*R);
  y = (int)random(2*R);
  pointsInSquare = pointsInSquare + 1;
  if (dist(x,y,R,R) < R) {
      pointsInCircle = pointsInCircle + 1;
      stroke(255,0,0);  // red
  }
  else
      stroke(0,0,255);  // blue
  point(x,y);
  println("PI estimated to: " + (double)pointsInCircle/pointsInSquare*4);
}
```

Notice that the code is very similar to the pseudocode. The **random(2*R)** function returns a random number from **0.0** to **2R-1**. This is a floating point number, and so we need to typecast to (**int**) in order to store them in variables **x** and **y**. Alternatively, we could have set the types of **x** and **y** to be **float** instead of **int**. The **dist(x,y,R,R)** function is a pre-defined routine that computes and returns the distance (in pixels) from point **(x,y)** to **point (R,R)**. Notice as well the use of (double) in the code. This ensures that the calculations are done as doubles, and not as integers, otherwise the result would always be **0**.

Each time through the draw loop, the code adds one new point It will take a long time for this code to produce a reasonable approximation for PI. We could add a **for** loop in the **draw()** procedure to add many points each time such as 100 or a 1000 … in order to speed up the approximation process.

Here is the adjusted code.

```
double  pointsInCircle = 0;
double  pointsInSquare = 0;
int     R = 250;

void setup() {
  size(2*R,2*R);
}

void draw() {
  float x, y;

  for (int i=0; i<1000; i++) {
    x = random(width);
    y = random(height);
    pointsInSquare = pointsInSquare + 1;
    if (dist(x,y,R,R) < R) {
      pointsInCircle = pointsInCircle + 1;
      stroke(255,0,0);  // red
    }
    else
      stroke(0,0,255);  // blue
    point(x,y);
  }

  println("PI estimated to: " + pointsInCircle/pointsInSquare*4);
}
```

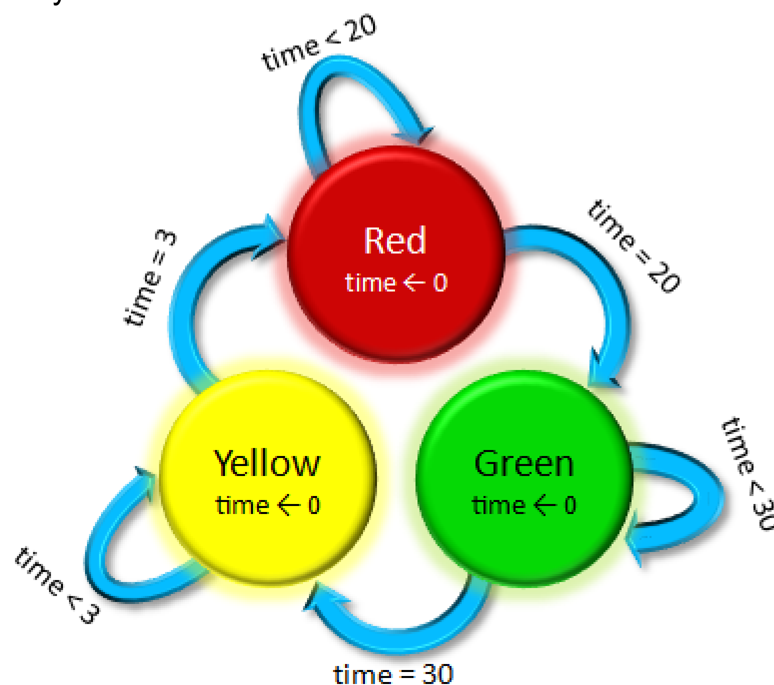
Example:

In the real world we objects have what is known as ***state***. The state of an object is normally considered to be some condition of the object with respect to a previous state. For example, a light bulb is considered to be in a "*working*" state when we buy it, but if we smash it on the ground it would then be in a "*broken*" state.

How can we simulate a traffic light ? It should have 3 states ... RED, GREEN and YELLOW. Assume that the traffic light starts in a RED state and that we want it to cycle continuously between these states. We will assume that the light remains RED for 20 seconds, then GREEN for 30 seconds, then YELLOW for 3 seconds. To simulate the traffic light, it is good to think of it as a state machine.

> *A* ***state machine*** *is any device that stores the status of something at a given time and can operate on input to change the status and/or cause an action or output to take place for any given change.*

We can then draw a ***state diagram*** to show how the traffic light changes from one state to another as time goes by:



Notice how the state changes from RED to GREEN only when the time has reached 20 seconds. Note as well that inside the state of GREEN, we reset the time counter to 0 so that we can count 30 seconds again in order to decide when to switch to the yellow state.

Assuming that we store the state of the traffic light as a string, how can we write the pseudocode to simulate the light ?

Algorithm: TrafficLight

```
1.     state ← "Red
2.     elapsedTime ← 0
3.     repeat {
4.         if (state is "Red") and (elapsedTime is 20) then {
5.             state ← "Green"
6.             elapsedTime ← 0
           }
7.         otherwise  if (state is "Green") and (elapsedTime is 30) then {
8.             state ← "Yellow"
9.             elapsedTime ← 0
           }
10.        otherwise  if (state is "Yellow") and (elapsedTime is 3) then {
11.            state ← "Red"
12.            elapsedTime ← 0
           }
13.        elapsedTime ← elapsedTime + 1
       }
```

The code correctly counts properly-proportioned time for each state of the traffic light. However, on a real computer, the **repeat** loop would run much faster than once per second. We would need to slow the whole simulation down so that the elapsed time increases only once per second. In Processing, the **repeat** loop would be represented by the **draw()** procedure which repeats indefinitely. The **frameRate(1)** function sets the re-draw rate of the draw() procedure so that it gets called once per second. This will allow the traffic light to operate at the correct speed.

```
String  state;// either red, green or yellow
int     time; // time elapsed since state changed

void setup() {
  state = "Red";
  time = 0;
  frameRate(4);
}

void draw() {
  if ((state == "Red") && (time == 20)) {
    state = "Green";
    time = 0;
    println(state);
  }
  else if ((state == "Green") && (time == 30)) {
    state = "Yellow";
    time = 0;
    println(state);
  }
  else if ((state == "Yellow") && (time == 3)) {
    state = "Red";
    time = 0;
    println(state);
  }
  time++;
}
```

For some fun, try writing code to draw the traffic light as it changes state.

3.2 Simulating Motion

For visually-appealing simulations, it is often necessary to show one or more objects moving on the screen. Such is certainly the case in the area of game programming. We will discuss here some code for doing very simple motion.

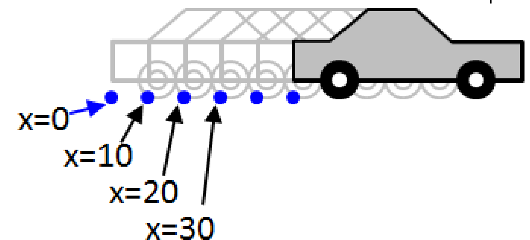
Example:

Recall the algorithm that we wrote for moving a car across the screen:

Algorithm: DrawCar
windowWidth: width of the window

1. **for** successive **x** locations from **0** to **windowWidth** {
2. draw the car at position **x**
3. **x** ← **x** + 10

}



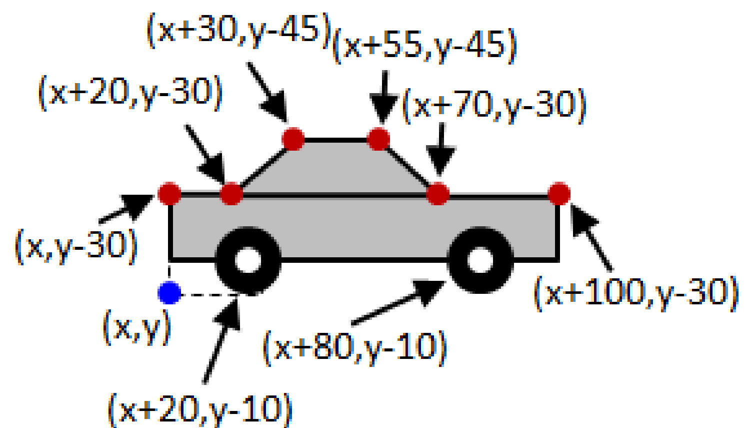
To do this in processing, the code would look like this:

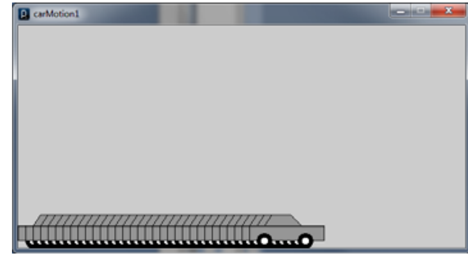
```
int  x, y;

void setup() {
  size(600,300);
  x = 0;
  y = 300;
}

void draw() {
  background(255,255,255);
  drawCar(x);
  x = x + 10;
}

void drawCar(int x) {
  // Draw the body
  fill(150,150,150);  // gray
  rect(x, y-30, 100, 20);
  quad(x+20,y-30,x+30,y-45,x+55,y-45,x+70,y-30);
  // Draw the wheels
  fill(0,0,0);  // black
  ellipse(x+20, y-10, 20, 20);
  ellipse(x+75, y-10, 20, 20);
  fill(255,255,255); // white
  ellipse(x+20, y-10, 10, 10);
  ellipse(x+75, y-10, 10, 10);
}
```





The code above shows the car travelling from the left side of the screen to the right. Notice that the **background()** procedure is called in order to erase the background (makes it white) before drawing the car each time. If we did not do this step, then the previous car positions would be visible.

The code above, however, does not stop the car at the edge of the screen. In order to stop the car, we need to do one of two things:

1) either stop changing the **x** value so that the car is redrawn at the same spot
2) stop the looping

The first solution is easy. We just need to change the last line of the **draw()** procedure to check if we have reached the end and only update when we are not there yet:

```
if (x+100 < width)
    x = x + 10;
```

Notice that we check for the position of the front bumper of the car (i.e**., x + 100**), not the back bumper (i.e., **x**). Recall as well that **width** is a pre-defined variable that represents the width of the window.

The other option is to stop the loop. There are two ways. First, we can simply call the **exit()** procedure within the draw() procedure in order to quit the program. However, this will cause our program to stop and the window will close. The 2nd way is to call **noLoop()** which temporarily disables the **draw()** procedure until **loop()** is called that will begin calling the **draw()** procedure again:

```
void draw() {
   background(255,255,255);
   drawCar(x);
   x = x + 10;
   if (x+100 > width)
      exit();
}
```

```
void draw() {
   background(255,255,255);
   drawCar(x);
   x = x + 10;
   if (x+100 > width)
      noLoop();
}
```

Example:

The above example showed our car moving rather quickly across the screen. If we adjusted the increment from 10 to a smaller value, the car would move much slower.

Recall our algorithm for accelerating the car until it reaches the middle of the window and then decelerating until it reached the right side of the window again:

```
1.      speed ← 0
2.      for x locations from 0 to windowWidth by speed {
3.          draw the car at position x
4.          if x < (windowWidth /2)
5.              speed ← speed + 0.10
6.          otherwise
7.              speed ← speed – 0.10
        }
```

It is quite easy to adjust our previous code to accomplish this:

```
int         x, y;
float       speed;

void setup() {
  size(600,300);
  x = 0;
  y = 300;
  speed = 0;
}

void draw() {
  background(255,255,255);
  drawCar(x);
  x = int(x + speed);
  if (x+100 > width)
     noLoop();
  if (x < width/2)
     speed = speed + 0.10;
  else
     speed = speed - 0.10;
}

void drawCar(int x) {
  // Draw the body
  fill(150,150,150);  // gray
  rect(x, y-30, 100, 20);
  quad(x+20,y-30,x+30,y-45,x+55,y-45,x+70,y-30);

  // Draw the wheels
  fill(0,0,0);  // black
  ellipse(x+20, y-10, 20, 20);
  ellipse(x+75, y-10, 20, 20);
  fill(255,255,255); // white
  ellipse(x+20, y-10, 10, 10);
  ellipse(x+75, y-10, 10, 10);
}
```

If we were to run the code, it seems as though the car speeds up, then slows down, but then it seems to crash (i.e., stop abruptly) at the end of the window. That is because we are using the back bumper of the car (i.e., position **x**) to decide when to speed up or slow down. Since we split the window ½ way for accelerating and decelerating, then the car will only reach a speed of 0 again when the back bumper reaches the end of the window. However, we stop the car before that (i.e., when the front bumper reaches the window's end).

To fix this, we simply need to adjust the **if** statement to accelerate/decelerate based on the center of the car as follows:

```
if ((x+50) < width/2)
   speed = speed + 0.10;
else
   speed = speed - 0.10;
```

If you make the change and run the code again, you will notice that the car oscillates back and forth now! Do you know why ?

Well, if you were to print out the **speed** value, you would notice that it goes from **0** to **7.5** and then back down to **0** … but then the value continues going backwards to **-6.6**!! So, the speed becomes negative. When we add this negative speed to the **x** value, it reduces the **x** value, making the car move backwards. So, how do we fix it ?

We just need to ensure that the speed never becomes negative. There are a couple of ways to do this … by adding one of these to the end of the **draw()** procedure:

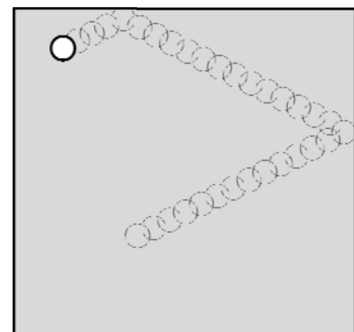
```
if (speed < 0)
    speed = 0;
```

```
speed = max(0, speed);
```

Example:

Now what about 2-dimensional motion ? How could we get a ball to bounce around the window so that it remains within the window borders ? To do this, we must understand the computational model.

To keep things simpler, lets assume that the ball is moving at a constant speed at all times. As the ball moves, we know that both its **x** and **y** locations will change. Also, the direction that the ball is facing should change. But when does the ball's direction change ? We will assume that it only changes direction when it hits the window borders.



So, we will need to keep track of the **ball's (x,y)** *location* as well as the *direction* (i.e., **angle**).

As with our moving car, we simply need to keep updating the ball's location and check to see whether or not it reaches the window borders. Here is the basic idea:

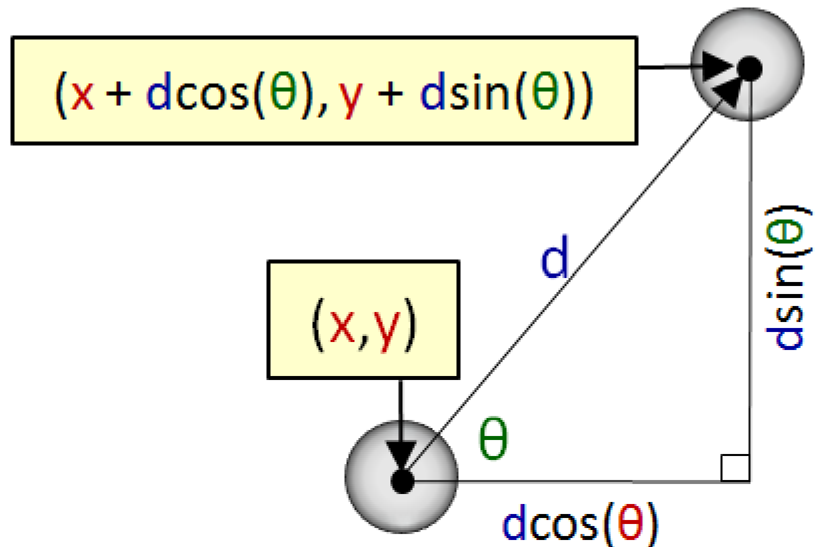
```
Algorithm: BouncingBall

1.      (x, y) ← center of the window
2.      direction ← a random angle from 0 to 2π
3.      repeat {
4.           draw the ball at position (x, y)
5.           move ball forward in its current direction
6.           if ((x, y) is beyond the window border) then
7.               change the direction accordingly
        }
```

It seems fairly straight forward, but two questions arise:

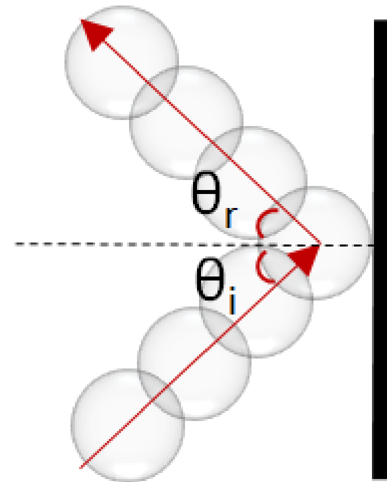
1) How do we "*move the ball forward in its current direction*" ?
2) How do we "*change the direction accordingly*" ?

The first is relatively simple, since it is just based on the trigonometry that we discussed it in the previous chapter. Given that the ball at location **(x,y)** travels distance **d** in direction **θ**, the ball moves an amount of **d•*cos*(θ)** horizontally and **d•*sin*(θ)** vertically as shown in the diagram. So, to get the new location, we simply add the horizontal component to x and the vertical component to y to get **(x + d*cos*(θ) , y + d*sin*(θ))**. Line 5 in the above algorithm therefore can be replaced by this more specific code (assuming that the ball moves at a speed of 10 pixels per iteration):



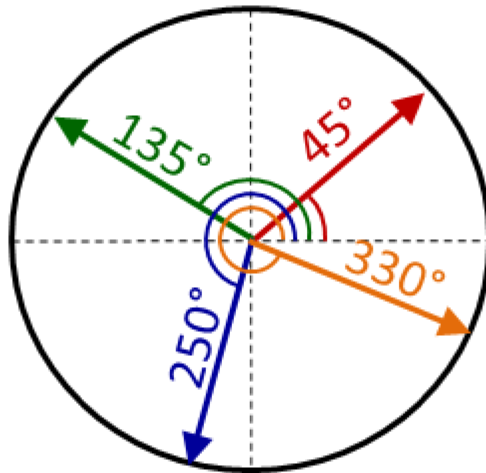

```
x ← x + 10 * cos(direction)
y ← y + 10 * sin(direction)
```

Now what about changing the direction when the ball encounters a window "wall" ? Well, we would probably like to simulate a realistic collision. To do this, we must understand what happens to a real ball when it hits a wall.



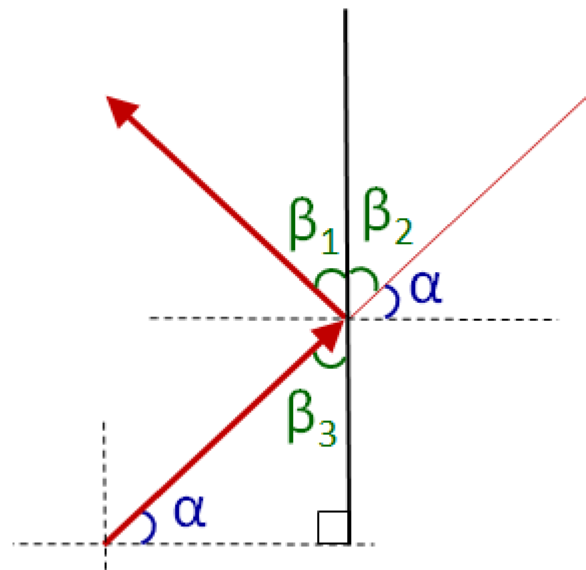
You may recall the ***law of reflection*** from science/physics class. It is often used to explain how light reflects off of a mirror. The law states that the ***angle of reflection*** is the same as the ***angle of incidence***, under ideal conditions. That is, the angle at which the ball bounces off the wall (i.e., θ_r in the diagram), will be the same as the angle at which it hit the wall (i.e., θ_i in the diagram).

However, where do we get the angle of incidence from ? Well, we have the direction of the ball stored in our **direction** variable.



This direction will always be an angle from **0** to **360°** (or from **0** to **2π** radians).

So, our ball's direction (called **α** for the purpose of this discussion) is always defined with respect to **0°** being the horizontal vector facing to the right. **360°** is the same as **0°**. As the direction changes counter-clockwise, the angle will increase. If the direction changes clockwise, the angle decreases. It is also possible that an angle can become negative. This is ok, since **330°** is the same as **-30°**.



Now, if you think back to the various angle theorems that you encountered in your math courses, you may remember these two:

1) the opposite angles of two straight crossing lines are equal

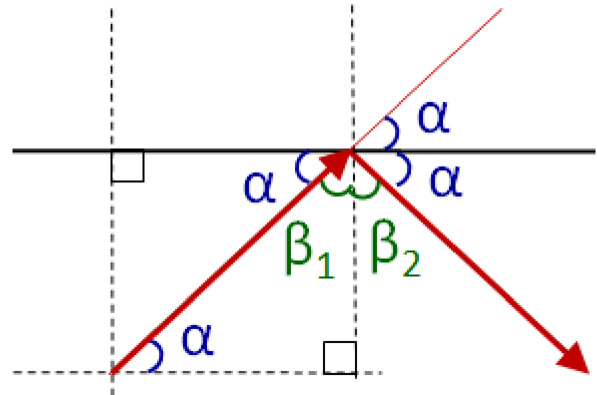
2) the interior angles of a triangle add up to **180°**

So, in the diagram on the right, for example, the 1st theorem above tells us that opposite angles $\boldsymbol{\beta_2}$ and $\boldsymbol{\beta_3}$ are equal. From the law of reflection, we also know that $\boldsymbol{\beta_1}$ and $\boldsymbol{\beta_3}$ are equal. Finally, **α** and $\boldsymbol{\beta_3}$ add up to **90°**.

What does all this mean ? Well, since α is the ball's direction, then to reflect off the wall, we simply need to add $\boldsymbol{\beta_1}$ and $\boldsymbol{\beta_2}$ to rotate the direction counter-clockwise. And since $\boldsymbol{\beta_1}$, $\boldsymbol{\beta_2}$ and $\boldsymbol{\beta_3}$ are all equal … and equal to **90° - α**, then to have the ball reflect we just need to do this:

direction = **direction** + (β_1 + β_2)
= **direction** + (90° - α + 90° - α)
= **direction** + (180° - 2 x **direction**)
= 180° - **direction**

The vertical bounce reflection is similar. In the diagram here, it is easy to see that β_1 **= 90° - α**. To adjust for the collision on the top of the window, we simply need to subtract **2α** from the direction:



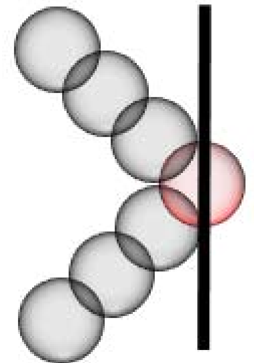
direction = **direction** - 2 x **direction**)
= **- direction**

To summarize then, when the ball reaches the left or right boundaries of the window, we negate the direction and add **180°**, but when it reaches the top or bottom boundaries, we just negate the direction. Here is how we do it:

```
Algorithm: BouncingBall
        windowWidth, windowHeight:          dimensions of the window

1.      x ← windowWidth/2
2.      y ← windowHeight/2
3.      direction ← a random angle from 0 to 2π
4.      repeat {
5.          draw the ball at position (x, y)
6.          x ← x + 10 * cos(direction)
7.          y ← y + 10 * sin(direction)
8.          if ((x >= windowWidth) OR (x <= 0)) then
9.              direction = 180° - direction
10.         if ((y >= windowHeight) OR (y <= 0)) then
11.             direction = - direction
        }
```

Our calculations made the assumption that the window boundaries are horizontal and vertical. Similar (yet more complex) formulas can be used for the case where the ball bounces off walls that are placed at some arbitrary angle. Also, all of our calculations assumed that the ball was a point. In reality though, the ball has a shape. If, for example, the ball was drawn as a circle centered at **(x,y)**, then it would only detect a collision when the center of the ball reached the border.



How could we fix this ?

We just need to account for the ball's radius during our collision checks:

```
if (((x+radius) >= windowWidth) OR (x-radius)  <= 0)) then
        ...
if ((y+radius)  >= windowHeight) OR (y-radius)  <= 0)) then
        ...
```

In processing, the code follows directly from, the pseudocode:

```
int             x, y;           // location of the ball at any time
float           direction;      // direction of the ball at any time

static float    SPEED = 10;     // the ball's speed
static int      RADIUS = 15;    // the ball's radius

void setup() {
  size(600,600);
  x = width/2;
  y = height/2;
  direction = random(TWO_PI);
}

void draw() {
  // erase the last ball position and draw the ball again
  background(0,0,0);
  ellipse(x, y, 2*RADIUS,2*RADIUS);

  // move the ball forward
  x = x + int(SPEED*cos(direction));
  y = y + int(SPEED*sin(direction));

  // check if ball collides with borders and adjust accordingly
  if ((x+RADIUS >= width) || (x-RADIUS <= 0))
    direction = PI - direction;
  if ((y+RADIUS >= height) || (y-RADIUS <= 0))
    direction = -direction;
}
```

Notice that the angles are in radians, instead of degrees. That is because the trigonometric functions **cos()** and **sin()** require angles in radians. Just for reference, **TWO_PI** and **PI** are constants defined in Processing that represent **2π** (i.e., **0°** or **360°**) and **π** (i.e., **180°**)

3.3 Event Handling

In many simulations, especially games, it is important to interact with the user. Sometimes the interaction is in regards to setting various simulation parameters before the simulation begins (i.e., wind speed, sensing diameter, start/end times, etc.). For gaming, there are other appropriate parameters such as difficulty settings, level of detail, game level, etc..

However, it is often necessary to interact with the user during the simulation by the means of buttons being pressed on the window, data being entered through dialog boxes, mouse clicks and keyboard presses, etc..

A problem arises, however, when dealing with such events. The problem is that the program is usually busy processing and displaying data in an endless loop. Since typical computers have only one processor (ignore multi-core for now), they can only literally do one thing at a time. However, the operating system of the computer is set up to handle user interaction ***events*** that arise while the computer is running.

> *An **event** is something that happens in the program based on some kind of triggering input which is typically caused (i.e., **generated**) by user interaction such as pressing a key on the keyboard, moving the mouse, or pressing a mouse button.*

These are called low-level events because they deal directly with physical interaction with the user. There are higher-level events that differentiate what the user is actually tying to do. For example, in JAVA, there are window-related events that get generated when the user clicks on a button, enters text in a text field or selects something from a list, etc. Regardless of the type of event that occurs, the programmer can decide what to do when these events occur by writing event handling routines:

> *An **event handler** is a procedure that specifies the code to be executed when a specific type of event occurs in the program.*

Typically, when dealing with user-interaction in an application, a programmer will write many event handlers, each corresponding to unique types of events. In fact, the event handlers may have different code to evaluate depending on the context of the program. For example, in a game, a mouse click may cause the game character to shoot a weapon when in one "mode" of the game, but when in another mode, the same mouse click may simply allow objects to be selected and moved around.

Some programs rely solely on events to determine what they will do. These are known as ***event-driven programs***. After initializing some values and perhaps opening a window to display something, event-driven programs enter into an infinite loop, called an ***event loop***.

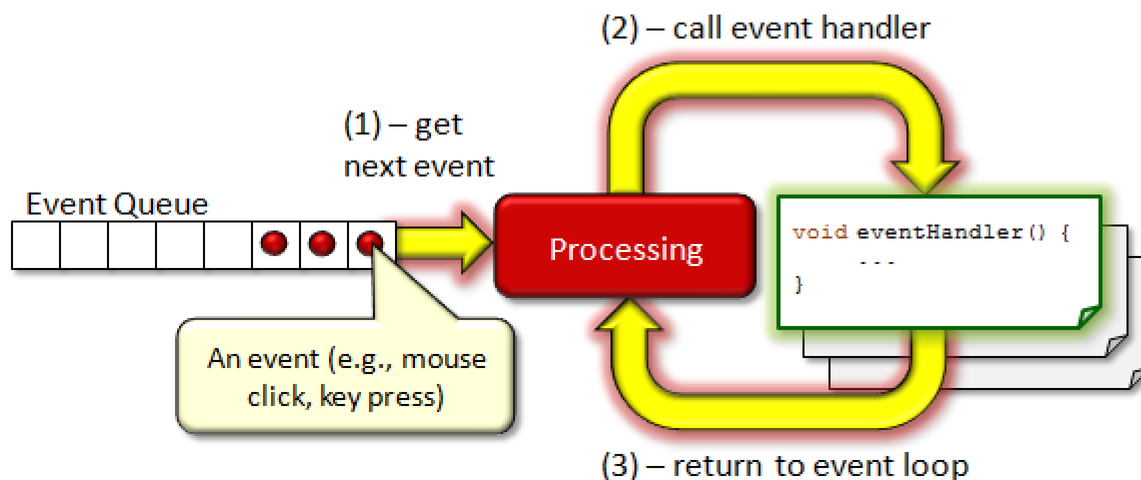
*An **event loop** is an infinite loop that waits for events to occur. When an event arrives, it is handled and then the program returns to the loop to wait for the next event.*

The idea of an event loop is similar to the notion of a store clerk waiting for customers ... the clerk does nothing unless an "event" occurs. Here are some events which may occur, along with how they may be handled:

- a customer arrives – store clerk wakes up and looks sharp
- a customer asks a question - store clerk gives an answer
- a customer goes to the cash to buy - store clerk handles the sale
- time becomes 6:00pm - store clerk goes home

When multiple customers arrive at the same time, only one can be served, so the others wait in line. Once a customer has been served, the next customer in line is served. This repeats until there are no more customers, in which case the store clerk waits patiently again for more customers or "events".

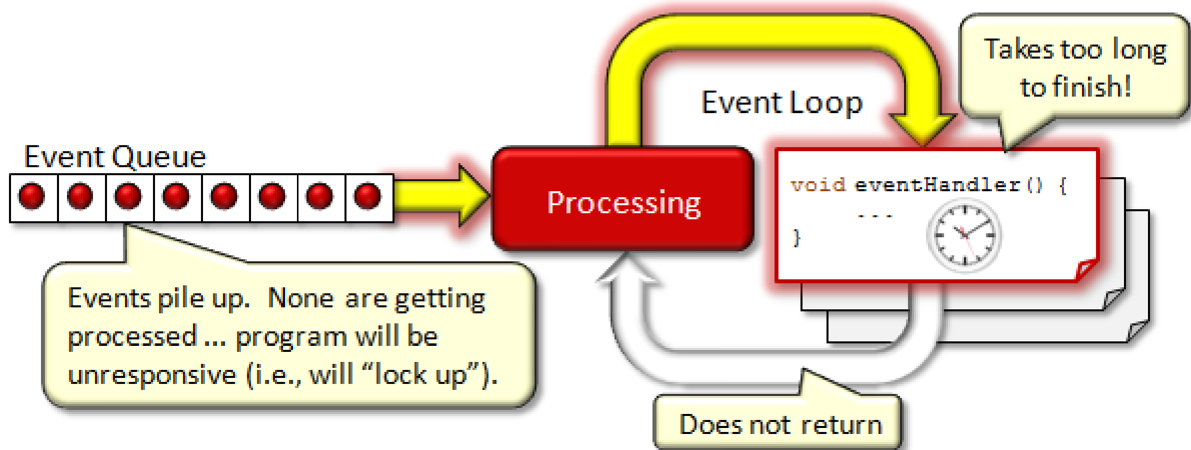
This line-up of customers is similar to what happens with the event loop. There is an ***event queue*** which is a "line-up" of incoming events that need to be handled (e.g., mouse click, key press, mouse move, etc…). Once an event has been handled to completion, the event loop extracts the next event from the queue and handles it:



Events are handled one at a time on a first-come-first-served basis. Event-driven programs continually operate in this manner. Often, the event queue is empty. In a typical application, the time between mouse clicks and keyboard presses is so large, that it is rare to have more than one or two events in the queue at any time. However, when a timer is used, it may generate events very quickly (e.g., once per millisecond) and the event queue can fill up quickly if the event handler is too slow.

For example, assume that your program sent an email when you clicked on a "Send" button. The clicking on the button is an event which will call an event-handling procedure. Assume that the event handler attempts to send the email. If it takes too long (as it usually does) to send the email then the event handler will not return right way. Meanwhile, the user may be

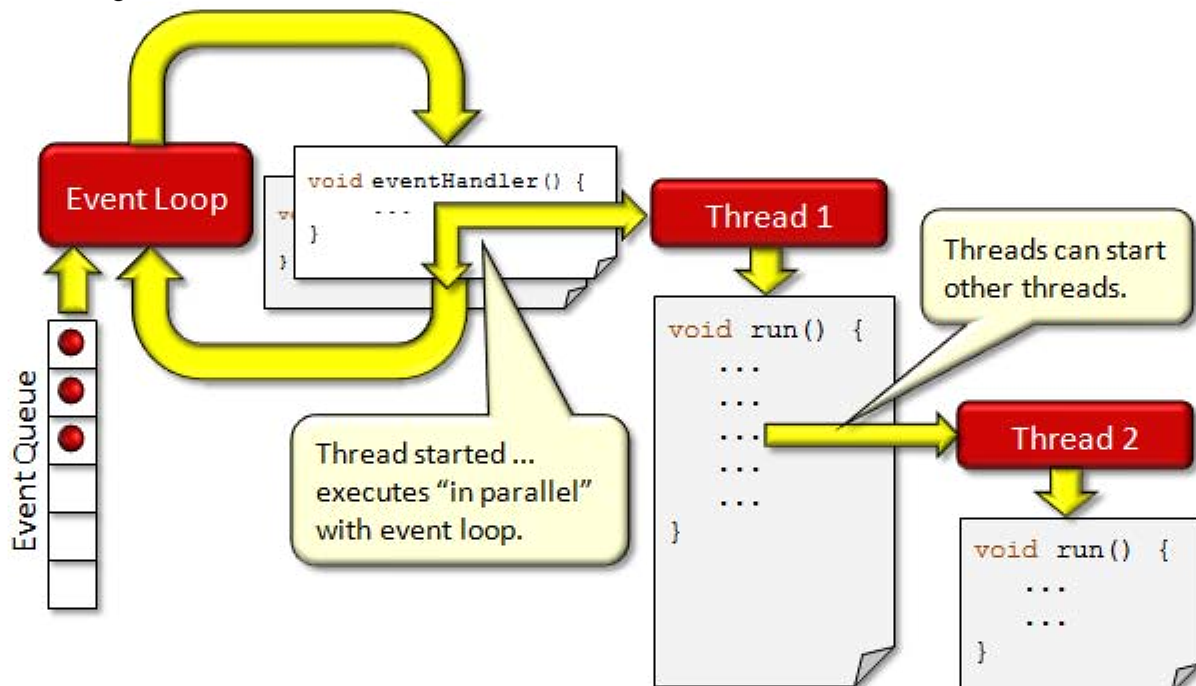
trying to click on various places on the window … but with no response. The window itself cannot be closed. What is happening is that when the user clicks, moves the mouse, presses a key etc.., these events are simply placed into the queue. However, the event loop is not taking any new events from the queue since it has not returned from the previous event. So the system has locked up until the "send email" event handler has returned.



So, it is important to make sure that event handling procedures do not take up too much time. If there is no choice in that the event handler may take a long time, then we can get around this "program lock-up" by spawning (i.e., starting up) what is called a ***thread***.

> *A **thread** (a.k.a. **process**) is a unit of processing that performs some tasks. It generally results from a fork (i.e., split) of a program into two or more tasks that run at the same time.*

The diagram below describes how a thread is started:



As shown in the diagram, a thread (i.e., new task) is started, perhaps from an event handling routine or from another thread. This thread acts just like a separate program (although it shares the same variable space as the program that it started from (i.e., was spawned from). The new thread continues on evaluating program code concurrently (i.e., at the same time as) the code that spawned it. So, we could, for example, spawn a thread to go send that email which takes a long time, while the original program continues on back to the event loop to handle more events. The newly spawned thread may simply stop when it has completed, or it too may loop repeatedly, performing other program-related tasks .. and it too can spawn additional threads at any time.

Although creating and starting threads is easy, writing programs that have multiple inter-communicating threads can be difficult. There are many timing-related issues that arise as well as the area of resource management. Programming with many threads/processes falls into the computer science areas of concurrent programming and parallel & distributed computing. We will not discuss this any further in the course.

3.4 User Interaction in Processing

Recall that in processing, the main event-handling loop is hidden. However, it allows you to write your own event handling procedures for some pre-defined types of events. Also, processing has a kind of "internal event handler" that sets the values of some useful variables which are related to the mouse and keyboard events. Here is a table of the mouse-related variables that are automatically set:

Variable	Description
mouseX	The current **x** (i.e., horizontal) position of the mouse
mouseY	The current **y** (i.e., vertical) position of the mouse
pmouseX	The previous **x** (i.e., horizontal) position of the mouse
pmouseY	The previous **y** (i.e., vertical) position of the mouse
mousePressed	**true** if the mouse button is being pressed, otherwise **false**
mouseButton	The mouse button that is being pressed (i.e., always one of **LEFT**, **RIGHT** or **CENTER**).

We can make use of these variables if we want our program to interact with the user based on the mouse movements and keyboard keys that are pressed.

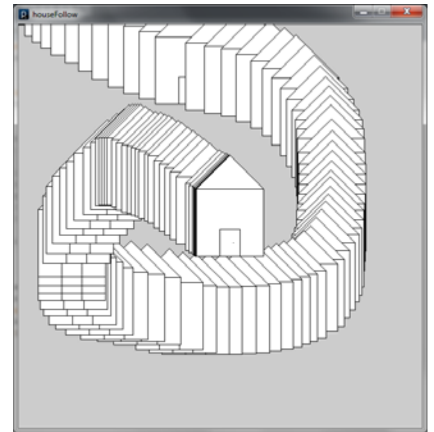
Example:

The following code will draw a house with its bottom-left at the mouse's current location:

```
void setup() {
     size(600,600);
}

void draw() {
     drawHouse(mouseX, mouseY);
}

void drawHouse(int x, int y) {
     rect(x, y-100,100,100);
     triangle(x,y-100,(x+50),y-150,(x+100),y-100);
     rect(x+35,y-40,30,40);
     point(x+55,y-20);
}
```



Notice how the house is draw repeatedly at whatever position the mouse is at currently. How could we alter this code so that it erases the old house ? Here are 2 choices:

a) we can erase the background each time or
b) erase the house at its previous location before drawing the new one.

Here is solution a):

```
void draw() {
  background(200);    // light gray = 200
  drawHouse(mouseX, mouseY);
}
```

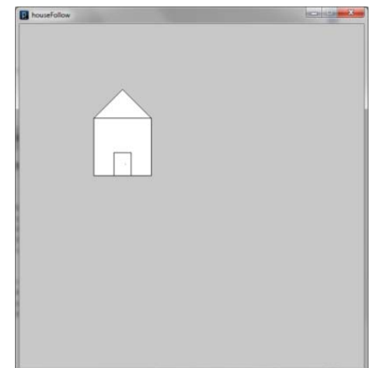
We just needed to add one line at the top of the **draw()** procedure that repaints the background … in this case gray with level 200 … a light gray.

Here is solution b):

```
void setup() {
     size(600,600);
     background(200);
}

void draw() {
     fill(200);
     stroke(200);
     drawHouse(pmouseX, pmouseY);
     fill(255);
     stroke(0);
     drawHouse(mouseX, mouseY);
}

void drawHouse(int x, int y) {} // same code as above… omitted to save space
```



Notice how it makes use of the previous mouse position (pmouseX, pmouseY) and "undraws" the house at that previous location. Some programming languages do not have such convenient variables. For example, if **pmouseX** and **pmouseY** were not available, how would you accomplish the same thing ? We would need to keep track of the previous position ourselves:

```
int  previousX, previousY;

void setup() {
    size(600,600);
    background(200);
    previousX = mouseX;
    previousY = mouseY;
}

void draw() {
    fill(200);
    stroke(200);
    drawHouse(previousX, previousY);
    fill(255);
    stroke(0);
    drawHouse(mouseX, mouseY);
    previousX = mouseX;
    previousY = mouseY;
}
```

Example:

How could we adjust the above code so that when the left mouse button is pressed, the house shrinks in scale but when the right mouse button is pressed, the house grows in scale ?

Well, to begin, we would need to recall our code that draws a house at a specific scale:

```
float scale = 1;

void drawHouse(int x, int y) {
  rect(x, y-100*scale,100*scale,100*scale);
  triangle(x,y-100*scale,(x+50*scale),
           y-150*scale,(x+100*scale),y-100*scale);
  rect((x+35*scale),y-40*scale,30*scale,40*scale);
  point((x+55*scale),y-20*scale);
}
```

We can then combine this code with some logic that increases or decreases the scale depending on the mouse button that was pressed. Here is the logic that will increase/decrease the scale by 1% each time:

if (left mouse button is being pressed) **then**
  scale ← **scale** * 0.99
if (right mouse button is being pressed) **then**
  scale ← **scale** * 1.01

However, in processing, the determination of whether or not a mouse button is being pressed is a separate function from the one that identifies which mouse button is being pressed. So, we cannot tell if the "left mouse button is being pressed" with one function call. First, we must determine whether or not a mouse button is being pressed by checking the **mousePressed** boolean variable. Then we may check which button is being pressed by checking the **mouseButton** variable as follows:

```
float scale;

void setup() {
     size(600,600);
     background(200);
     scale = 1;
}

void draw() {
     background(200);
     drawHouse(mouseX, mouseY);
     if (mousePressed) {
          if (mouseButton == RIGHT)
            scale = scale * 1.01;
          else
            scale = scale * 0.99;
     }
}

void drawHouse(int x, int y) {
  rect(x, y-100*scale,100*scale,100*scale);
  triangle(x,y-100*scale,(x+50*scale),
              y-150*scale,(x+100*scale),y-100*scale);
  rect((x+35*scale),y-40*scale,30*scale,40*scale);
  point((x+55*scale),y-20*scale);
}
```

Similarly, here is a table of the keyboard-related variables that are automatically set:

Variable	Description
keyPressed	**true** if a keyboard key is being pressed, otherwise **false**
key	A character representing the key that was most recently pressed (e.g., ‘**a**’, ‘**A**’, ‘**1**’, ‘**.**’, etc.., **BACKSPACE**, **TAB**, **ENTER**, **RETURN**, **ESC**, and **DELETE**).
keyCode	A constant representing a “special” key that was most recently pressed (i.e., **UP**, **DOWN**, **LEFT**, **RIGHT** arrow keys and **ALT**, **CONTROL**, **SHIFT**).

#

Example:

How would we adjust our above example code so that the house becomes darker when the **up arrow** key is pressed and lighter when the **down arrow** is pressed ?

We would need to add a **shade** variable to keep track of the current color of the house and adjust it accordingly:

```
shade ← 128
if (up arrow is being pressed) then
        shade ← shade + 1
if (down arrow is being pressed) then
        shade ← shade – 1
```

Of course, we would need to ensure that our **shade** always remains in the range of **0** to **255**:

```
if (up arrow is being pressed) AND (shade < 255) then
        shade ← shade + 1
if (down arrow is being pressed) AND (shade > 0)  then
        shade ← shade – 1
```

In processing, the code follows from this logic. We would first need to check to make sure that a key is being pressed using the **keyPressed** variable, and then (since the arrow keys are special keys) use **keyCode** to see whether or not it was the up or down arrow.

Here is the code:

```
float scale;
int   shade;

void setup() {
     // same code as before ... omitted to save space
     shade = 128;
}

void draw() {
     background(200);
     fill(shade);
     drawHouse(mouseX, mouseY);
     if (mousePressed) {}  // same code as before ... omitted to save space
     if (keyPressed) {
          if (key == CODED) { // Required before checking "special" keys
               if ((keyCode == UP) && (shade < 255))
                    shade = shade + 1;
               else if ((keyCode == DOWN) && (shade > 0))
                    shade = shade - 1;
          }
     }
}
void drawHouse(int x, int y) {} // same as before ... omitted to save space
```

You may have noticed that we checked to see whether or not the **key == CODED**. This is required whenever checking for special keys. If, for example we were checking for keys **'a'** and **'b'**, instead of **UP** and **DOWN**, the code would simply use the **key** variable as follows:

```
if (keyPressed) {
     if ((key == 'a') && (shade < 255))
          shade++;
     else if ((key == 'b') && (shade > 0))
          shade--;
}
```

The above examples show how to use some of the pre-defined variables in Processing. However, not all programming languages have such variables readily available. In JAVA, for example, if you want to access the mouse position or determine the key that was pressed,, it is necessary to write your own event handler.

Processing allows you to write event handlers for some predefined types of events. In order to have this event handler called automatically when the event occurs, it is important to "spell" the event handler exactly the way that Processing is expecting.

```
void eventHandler() {
     ...
}
```

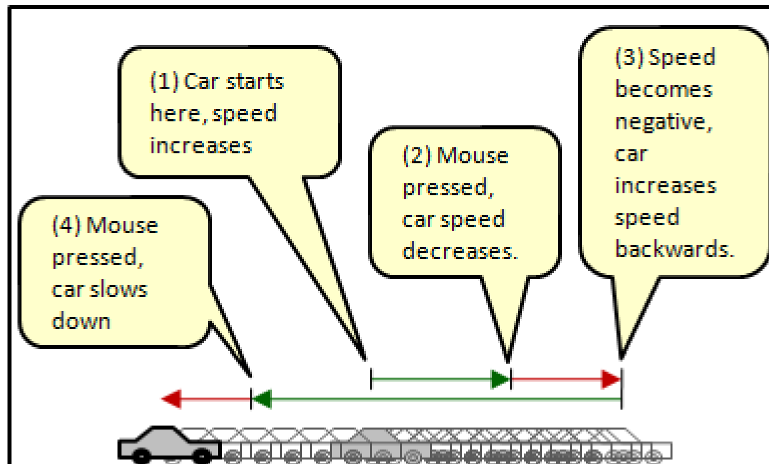
Below is a table showing which procedures need to written in order to handle some specific events:

Event Handling Procedure	Description
void mousePressed() { … }	called when a mouse button is pressed
void mouseReleased() { … }	called when a mouse button is released
void mouseClicked() { … }	called when a mouse button is pressed & released
void mouseMoved() { … }	called when the mouse is moved
void mouseDragged() { … }	called when the mouse is moved while a button is being pressed
void keyPressed() { … }	called when a key is pressed on the keyboard
void keyReleased() { … }	called when a key is released on the keyboard
void keyTyped() { … }	called when a key is pressed and released on the keyboard

Example:

Can we adjust the car program so that it accelerates horizontally towards the right but then when we press a mouse button, it should slow down and change directions, heading backwards to the left. Again, if we press a mouse button it should slow down and head right again. Therefore the car alternates from right to left all the while remaining within the window.

This is a good example of where an event handler could be used. The car should speed up at all times, only slowing down on a direction change (indicated by a mouse press event).



As usual, we need to understand the model first. Below is the sequence of speed values (assuming the speed starts at 0 and accelerating/decelerating with a value of 1) that would occur as the car is moving. Notice how the speed changes when the event occurs (note that this goes a bit beyond the car movement shown above):

mouse click																										
speed	0	1	2	3	4	5	4	3	2	1	0	-1	-2	-3	-4	-5	-4	-3	-2	-1	0	1	2	3	4	5
increment	1	1	1	1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	1	1	1	1
car direction	→	→	→	→	→	→	→	→	→	→		←	←	←	←	←	←	←	←	←		→	→	→	→	→

Notice how the speed increases by the increment value until the mouse is clicked. It then "increases" by a negative increment value (i.e., decreases) until the next mouse click. We can make use of an **acceleration** variable that it is positive when speeding up and negative when slowing down, corresponding to the **increment** value in the above table.

Recall our algorithm that caused the car to speed up until the car reached half way, then slowed down:

```
Algorithm: AcceleratingCar
        windowWidth:            width of the window

1.    speed ← 0
2.    repeat {
3.        draw the car at position x
4.        x ← x + speed
5.        if x < (windowWidth /2) then
6.            speed ← speed + 0.10
7.        otherwise
8.            speed ← speed – 0.10
      }
```

Our new algorithm will be similar except that we now need to incorporate the acceleration/deceleration and also adjust to ensure that the car does not go beyond the window boundaries:

```
Algorithm: AlternatingCar
        windowWidth:            width of the window

1.    speed ← 0
2.    acceleration ← 0.10
3.    repeat {
4.        draw the car at position x
5.        x ← x + speed
6.        speed ← speed + acceleration
7.        if x < 0 then {
8.            speed ← 0
9.            x ← 0
          }
10.       if (x +100) > windowWidth then {
11.           speed ← 0
12.           x ← windowWidth - 100
          }
     }
```

All that remains is to set up an event handler for when the mouse button is clicked. In that event handler, all we need to do is negate the acceleration:

acceleration ← **acceleration** * -1

That is all. Here is the corresponding Processing code:

```
int         x, y;
float       speed;
float       acceleration;

void setup() {
  size(600,300);
  x = 0;
  y = 300;
  speed = 0;
  acceleration = 0.10;
}

void draw() {
  background(255,255,255);
  drawCar(x);

  x = int(x + speed);
  speed = speed + acceleration;

  if (x < 0) {
    x = 0;
    speed = 0;
  }
  else if (x+100 > width) {
    x = width - 100;
    speed = 0;
  }
}

// This is the mouse pressed event handler
void mousePressed() {
  acceleration = acceleration * -1;
}

void drawCar(int x) {
  // … same code as before … omitted to save space.
}
```

Example:

Recall our bouncing ball example. Below is the adjusted code to ensure that the ball remains within the screen and with a speed variable added:

```
Algorithm: BouncingBall
        windowWidth, windowHeight:          dimensions of the window
        radius:                             the ball's radius

1.      x ← windowWidth/2
2.      y ← windowHeight/2
3.      direction ← a random angle from 0 to 2π
4.      speed ← 10
5.      repeat {
6.           draw the ball at position (x, y)
7.           x ← x + speed * cos(direction)
8.           y ← y + speed * sin(direction)
9.           if ((x+radius >= windowWidth) OR (x-radius <= 0)) then
10.              direction = 180° - direction
11.          if ((y+radius >= windowHeight) OR (y-radius <= 0)) then
12.              direction = - direction
        }
```

How can we adjust this code using event handlers so that the ball can be “grabbed” and “thrown” by the mouse ? That is, if the user places the mouse cursor over the ball and clicks, then the ball stops moving and appears to be “stuck” to the mouse cursor until the mouse is released. Then when we let go of the mouse button, the ball should “fly off” in the direction that we threw it with a speed that varies according to how “hard” we threw it.

To do this, we should break the problem down into more manageable steps:

1. Add the ability to grab the ball and carry it around
2. Add the ability to throw the ball
3. Adjust the speed of the ball according to how “hard” we threw it.

To grab the ball, we would need to prevent lines 7 and 8 from being evaluated while the ball is being held. Instead, we would set the ball’s location to be the mouse location.

We can create a boolean to determine whether or not the ball is being held:

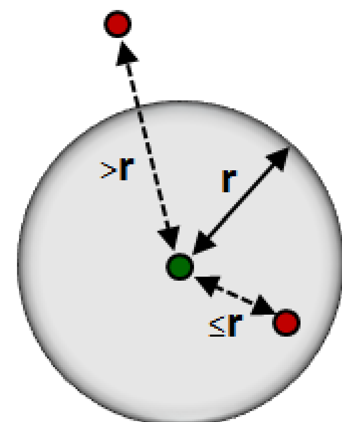
```
grabbed ← false
repeat {
    …
    if not grabbed then {
        x ← x + speed * cos(direction)
        y ← y + speed * sin(direction)
    }
    otherwise {
        x ← x position of mouse
        y ← y position of mouse
    }
    …
}
```

All that would be left to do is to set the **grabbed** variable accordingly. When the user would click on the ball, we should set it to **true** and when the user releases the mouse button, we should set it to **false**. So we need two event handlers:

```
mousePressed() {
    grabbed ← true
}
mouseReleased() {
    grabbed ← false
}
```

But this will ALWAYS "grab" the ball, even if the mouse cursor was not on it. How can we determine whether or not the mouse cursor is over the ball ? We can check to see whether or not the mouse location is within (i.e., **≤**) the ball's radius.

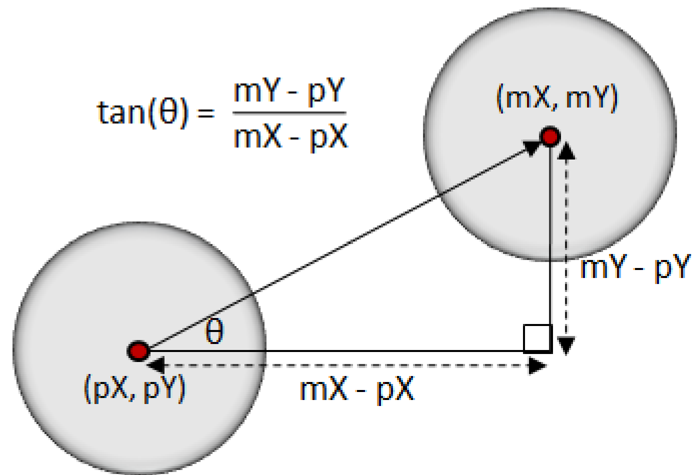
We can compute the distance from the ball's center (i.e., **(x,y)**) to the location of the mouse **(mX, mY)**. If this distance is less than or equal to the ball's radius, then we can assume that the user has "grabbed" the ball. Here is the adjusted code:




```
mousePressed() {
    d ← distance from (x, y) to (mX, mY)
    if (d <= radius) then
        grabbed ← true
}
```

When the user lets go of the ball, it will continue in the direction that it was in before it was grabbed. Now how do we adjust the code so that we are able to "throw" the ball in some particular direction ?

Well, upon releasing the mouse, we will need to determine which direction the ball was being thrown in and then set the **direction** variable directly. We can determine the direction that the ball was thrown in by examining the current mouse location **(mX, mY)** with respect to the previous mouse location **(pX, pY)**.



The angle (i.e., **θ**) at which the ball should be thrown will be the **arctangent** of the differences in **x** and **y** coordinates as shown here.

However, in the case that we throw vertically, the difference in **x** coordinates will be zero and we are not allowed to divide by zero. Fortunately, many computer languages have a function called **atan2(y, x)** which allows you to find the angle that a point makes with respect to the origin **(0,0)**. We can make use of this by assuming that **(pX,pY)** is the origin and translate **(mX,mY)** accordingly as follows: **atan2(mY-pY, mX-pX)**

So, upon a mouse release, we can do this:

```
mouseReleased() {
    if grabbed then {
        direction ← atan2(mY-pY, mX-pX)
        grabbed ← false
    }
}
```

Notice that we only change the direction when we have already grabbed the ball.

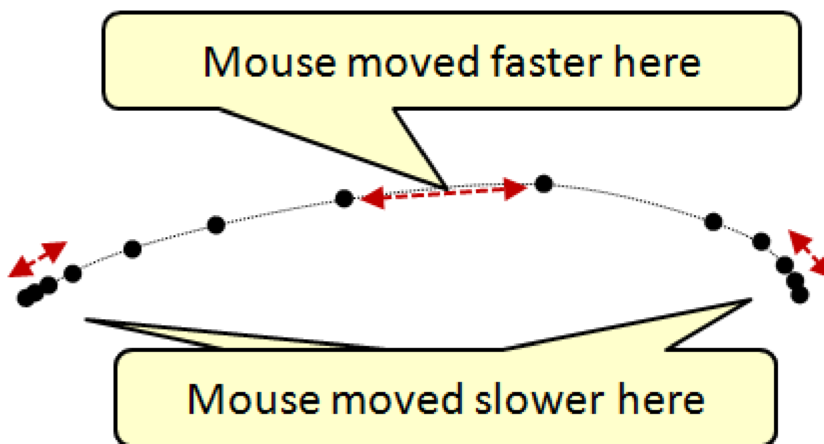
One last aspect of the program is to allow the ball to be thrown at various speeds. Likely, we want the ball to slow down as time goes on.

We should add the following to the algorithm's main **repeat** loop:

```
speed ← speed – 0.1
if (speed < 0) then
    speed ← 0
```

Now to determine the speed at which the ball is thrown, we can take notice of how the mouse location varies according to the speed at which it is moved.

If the mouse is moved fast, the successive locations of the mouse will be further apart, while slow mouse movements will have successive locations that are relatively closer together.



So, as a simple strategy, the amount of "throw hardness" can be computed as a function of the distance between the current mouse location and the previous mouse location.

We can simply set the **speed** to this distance in the **mouseReleased**() handler as follows:

```
mouseReleased() {
    if grabbed then {
        direction ← atan2(mY-pY, mX-pX)
        speed ← distance from (pX,pY) to (mX,mY)
        grabbed ← false
    }
}
```

Here is the resulting Processing code:

```
int         x, y;        // location of the ball at any time
float       direction;   // direction of the ball at any time
boolean     grabbed;     // true if the ball is being held
float       speed;       // the ball's speed

final int   RADIUS = 40;             // the ball's radius
final float ACCELERATION = 0.10;     // acceleration/deceleration amount

void setup() {
  size(600,600);
  x = width/2;
  y = height/2;
  direction = random(TWO_PI);
  grabbed = false;
  speed = 10;
}

void draw() {
  background(0,0,0);
  ellipse(x, y, 2*RADIUS,2*RADIUS);

  // move the ball forward if not being held
  if (!grabbed) {
    x = x + int(speed*cos(direction));
    y = y + int(speed*sin(direction));
  }
  else {
    x = mouseX;
    y = mouseY;
  }

  speed = max(0, speed - ACCELERATION);

  if ((x+RADIUS >= width) || (x-RADIUS <= 0))
    direction = PI - direction;
  if ((y+RADIUS >= height) || (y-RADIUS <= 0))
    direction = -direction;
}

void mousePressed() {
  if (dist(x,y,mouseX,mouseY) < RADIUS)
    grabbed = true;
}

void mouseReleased() {
  if (grabbed) {
    direction = atan2(mouseY - pmouseY, mouseX - pmouseX);
    speed = int(dist(mouseX, mouseY, pmouseX, pmouseY));
  }
  grabbed = false;
}
```