Event-Driven Programming

CPSC 231 - FALL 2018

This supplementary handout describes a program execution model that you might find useful for Assignment #5. Though our official course text does not formally cover this topic, the `stddraw` module does provide us with the means necessary to use this model.

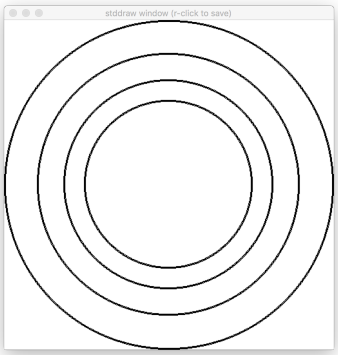
Up to now, the model of execution for the computer programs we've written have been entirely sequential. Program statements are executed one at a time, in a certain order, obeying your conditional and iterative statements, until the program terminates. The only interactions our programs have had with a human have been through the console via the `input()` or `stdio.read*()` functions. While it has served us well, the key limitation of this model is that *we cannot do anything while waiting for input from the human!*

For example, if we wanted to write a program that draws something until the user indicates otherwise, it might look like the listing below. Although you can interact with the program through the console, your operating system may start to get unhappy that your program is not being responsive. To some extent, it's right, because no modern application with a graphical interface works this way.[1]

[1] With Assignment #4, we were kind of at a "teething" stage in transitioning to a visual interface for our programs. It made the most sense to tolerate this awkwardness for just a little while.

sequential-circles.py

```
import stddraw
radius = 0.5
keep_drawing = True
while keep_drawing:
    stddraw.circle(0.5, 0.5, radius)
    stddraw.show(0.0)
    radius *= 0.8
    response = input('Draw another circle? ')
    keep_drawing = (response == 'y')
```



Output of running `sequential-circles.py`:

```
Draw another circle? y
Draw another circle? y
Draw another circle? y
Draw another circle? n
```

What if we wanted to run a live animation that responds to a mouse click? Or perhaps alter its behaviour if we press a certain key? The *event-driven* programming paradigm provides a means to achieve this. The main idea is to run an indefinite loop, called an *event loop*, that forms the core of your program. On each iteration, check if any user input *events* have occurred, respond accordingly, then take care of other things for a finite amount of time.

Our textbook's `stddraw` module provides a simple means for us to respond to mouse clicks in an event-driven manner. Calling `stddraw.mousePressed()` will return `True` to us if the mouse had been clicked since the last time we asked. Then we can obtain the position of the last mouse click, in our `stddraw` window coordinates, with `stddraw.mouseX()` and `stddraw.mouseY()`. Using these functions, we can turn the bouncing ball example (Program 1.5.6) into a crude "game" of trying to click on the ball, as shown.

The event-driven algorithm would look something like this:

- While we want to keep running our program, repeat:
  1. Check for input or other events.
  2. Respond to events that occurred.
  3. Do some other things for a small, finite amount of time.

Many windowing system APIs will run the event loop for you and check for events automatically, invoking specified functions in your code, called *event handlers*, whenever events you're interested in occur.

event-driven-ball.py

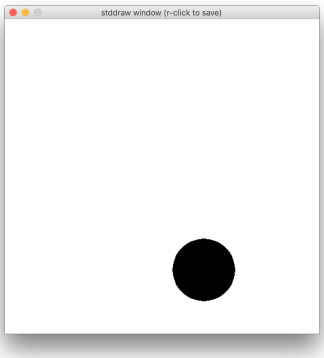
```
import stddraw

radius = 0.1
rx = 0.5    # ball x-position
ry = 0.5    # ball y-position
vx = 1.5    # ball x-velocity
vy = 2.3    # ball y-velocity
dt = 0.01   # time step for animation
clicked_on_ball = False

while not clicked_on_ball:
    # check for events and respond accordingly
    if stddraw.mousePressed():
        mx = stddraw.mouseX()
        my = stddraw.mouseY()
        d_squared = (mx-rx)*(mx-rx) + (my-ry)*(my-ry)
        clicked_on_ball = (d_squared < radius * radius)

    # animate the ball and bounce off walls
    rx += dt * vx
    ry += dt * vy
    if rx < 0.0 or rx > 1.0:
        vx *= -1.0
    if ry < 0.0 or ry > 1.0:
        vy *= -1.0

    stddraw.clear()
    stddraw.filledCircle(rx, ry, radius)
    stddraw.show(1000.0 * dt)
```



Similarly, `stddraw` provides a means to respond to key presses in an event-driven manner, without going through the console. Calling `stddraw.hasNextKeyTyped()` gives us `True` if a key was pressed since we last checked,[2] and `stddraw.nextKeyTyped()` gives us the key as a one-character `str` object. Try the frustrating little reflex-tester demonstration program listed below!

[2] Note that key presses are queued up in a buffer so that we don't lose keys if the user is typing faster than the rate at which you're running your event loop. The best way to process key events is to use a `while` statement like that shown in the example program.

event-driven-keys.py

```
import random
import stddraw

stddraw.setFontSize(300)
matched = False
while not matched:
    number = str(random.randrange(1, 10))
    stddraw.clear()
    stddraw.text(0.5, 0.5, number)
    stddraw.show(500.0)
    while stddraw.hasNextKeyTyped():
        key = stddraw.nextKeyTyped()
        matched = matched or (key == number)
```