Timer-Based Animation

Eric Roberts
CSCI 121
September 20, 2019

Timer Events

- The programs you saw on Wednesday responded to mouse events by adding an event listener to the `GWindow` object.
- Python also allows you to listen for timer events, which occur after a specified time interval.
- As with mouse events, you specify the listener for a timer event in the form of a callback function that is automatically invoked at the end of the time interval.
- You can add animation to the graphics window by creating a timer whose callback function makes small updates to the graphical objects in the window.
- If the time interval is short enough (typically between 20 and 30 milliseconds), the animation will appear smooth to the human eye.

One-Shot and Interval Timers

- The Portable Graphics Library supports two kinds of timers. A one-shot timer invokes its callback function once after a specified delay. You create a one-shot timer by calling
  
  ```java
  gw.setTimeout(function, delay)
  ```

  where `function` is the callback function and `delay` is the time interval in milliseconds.

- An interval timer invokes its callback function repeatedly at regular intervals. You create an interval timer by calling
  
  ```java
  gw setInterval(function, delay)
  ```

- The `setTimeout` and `setInterval` methods return a `GTimer` object that identifies the timer. You can stop the timer by invoking the `stop` method on that timer.

The AnimatedSquare Example

```python
def AnimatedSquare():
    def step():
        gw = GWindow(400, 400)
        dx = (gw.getWidth() - SQUARE_SIZE) / N_STEPS
        dy = (gw.getHeight() - SQUARE_SIZE) / N_STEPS
        square = createFilledRect(0, 0, SQUARE_SIZE, SQUARE_SIZE)
        gw.add(square)
        timer = gw setInterval(step, TIME_STEP)
    gw.addMouseListener(step)
    gw.addMouseMotionListener(step)
```

Simulations

- Defining the precise equations for an elliptical orbit requires mathematical techniques beyond the scope of CSCI 121.
- As is often the case, it is much easier to generate an orbital path using the technique of discrete simulation, in which you break down a complex sequence of events into short intervals in which it is safe to ignore much of the overall complexity.
- To simulate an orbit, for example, you can write a program that divides time up into a series of short time steps. In each time step, you perform the following operations:
  - Update the planet’s position based on its velocity.
  - Update the planet’s velocity based on the forces acting on it.
- If the time interval is short enough, you can ignore the fact that the position and velocity are changing continuously and not changing once in the time step.

Orbital Mechanics

- Planetary motion is controlled by initial velocity and gravity.

  - Kepler’s Second Law: Planets sweep out equal areas in equal time.
Structure of the Simulation

- As in Breakout, animating two-dimensional motion requires you to maintain separate x and y components for position, velocity, and acceleration, which are stored in the following variables:
  - \( ax, ay \): The position of the sun, which remains constant
  - \( px, py \): The position of the planet
  - \( vx, vy \): The velocity of the planet
  - \( ax, ay \): The acceleration that gravity imposes on the planet
- In each time step, the code performs the following actions:
  - Compute the acceleration, which follows an inverse-square law
  - Adjust the velocity by the components of the acceleration
  - Adjust the position by the components of the velocity

Simulating the Force of Gravity

- The distance between the sun and the planet can be computed from the coordinates using the Pythagorean Theorem:
  \[ r = \sqrt{(px - ax)^2 + (py - ay)^2} \]
- The acceleration due to gravity is given by the equation
  \[ a = \frac{G m_1 m_2}{r^2} \]
  where \( G \) is the gravitational constant, \( m_1 \) and \( m_2 \) are the masses of the two bodies, and \( r \) is the distance between them.
- The components \( ax \) and \( ay \) are in the same proportion as the differences in the coordinates.

Code for SimulateOrbit

```python
def SimulateOrbit():
    def step():
        gw = pygame.display.get_surface()
        ax = gw.get_width() / 2
        ay = gw.get_height() / 2
        sun = pygame.draw.circle(gw, SUN_RADIUS, "Goldenrod")
        gw.add_sprites()
        px = ax + INITIAL_X_DISTANCE
        py = ay
        planet = pygame.draw.circle(px, py, PLANET_RADIUS, "Blue")
        gw.add_sprites()
        vx = INITIAL_VEL
        vy = INITIAL_VEL
        timer = gw.setInterval(step, TIME_STEP)
```

Nondeterministic Programming

- The next several slides offer an introduction to the built-in `random` library in Python, which makes it possible to write programs that simulate random processes such as flipping a coin or rolling a die.
- Programs that involve random processes that cannot be predicted in advance are said to be nondeterministic.
- Nondeterministic behavior is essential to many applications. Computer games would cease to be fun if they behaved in exactly the same way each time. Nondeterminism also has important practical uses in simulations, in computer security, and in algorithmic research.

Functions in the `random` Library

| Random integers | `randint(min, max)` | Returns a random integer between \( \text{min} \) and \( \text{max} \), inclusive.
| Random integers | `randrange(\text{limit})` | Returns a random integer from 0 up to but not including \( \text{limit} \).
| Random integers | `randrange(\text{start}, \text{limit})` | Returns an integer from \( \text{start} \) up to but not including \( \text{limit} \).
| Random floating-point numbers | `random()` | Returns a random number between 0 and 1.
| Random floating-point numbers | `uniform(min, max)` | Returns a random number between \( \text{min} \) and \( \text{max} \).
| Random functions on lists | `choice(list)` | Returns a random element from \( \text{list} \).
| Random functions on lists | `sample(list, \text{k})` | Returns a list of \( \text{k} \) elements randomly chosen from \( \text{list} \).
| Random functions on lists | `shuffle(list)` | Randomly shuffles the elements of \( \text{list} \).
| Initialization functions | `seed()` | Initializes the internal random number generator.
| Initialization functions | `seed(\text{dict})` | Sets a repeatable starting point based on the integer \( \text{dict} \).
Exercises: Generating Random Values
How would you solve each of the following problems?

1. Set the variable `total` to the sum of two six-sided dice.
   ```python
   total = randomInteger(2, 12);
   ```

2. Set the variable `x` to a random number between –1 and +1.

3. Randomly set the variable `flip` to “Heads” or “Tails”.

Example: Random Circles
- The `RandomCircles` program on page 136 draws a set of ten circles with random sizes, positions, and colors, subject to the condition that the circles must fit inside the window.

Code for `RandomCircles.py`
```python
# File: RandomCircles.py
...
from pgl import GWindow, Goval
from gnumpy import createFilledCircle
import random

# Constants
GWINDOW_WIDTH = 500
GWINDOW_HEIGHT = 300
N_CIRCLES = 10
MIN_RADIUS = 10
MAX_RADIUS = 50

# Main program
def RandomCircles():
    random.seed()
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    for i in range(N_CIRCLES):
        gw.add(createFilledCircle())

def createFilledCircle():
    # Creates a randomly generated circle that fits in the window
    x = random.uniform(MIN_RADIUS, MAX_RADIUS)
    y = random.uniform(0, GWINDOW_HEIGHT - y)
    return createFilledCircle(x, y, randomColor())

# Startup code
if __name__ == "__main__":
    RandomCircles()
```

Code for `RandomCircles.py`
```python
# Code for RandomCircles.py

def randomColor():
    # Returns a random opaque color expressed as a string
    # consisting of a # followed by six random hexadecimal digits.
    s = "#" + ''.join([random.choice('0123456789ABCDEF') for i in range(4)])
    return s

# Main program
def RandomCircles():
    random.seed()
    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    for i in range(N_CIRCLES):
        gw.add(createFilledCircle())

def createFilledCircle():
    # Creates a randomly generated circle that fits in the window
    x = random.uniform(MIN_RADIUS, MAX_RADIUS)
    y = random.uniform(r, GWINDOW_HEIGHT - y)
    return createFilledCircle(x, y, randomColor())
```

Code for `RandomCircles.py`
```python

def randomColor():
    # Returns a random opaque color expressed as a string
    # consisting of a # followed by six random hexadecimal digits.
    str = "#" + ''.join([random.choice('0123456789ABCDEF') for i in range(4)])
    return str

# Startup code
if __name__ == "__main__":
    RandomCircles()

```

Graphics Contest Gallery
- The CSCI 121 Graphics Contest
- Winners’ Gallery