Simulation – Events and Randomness

15-110 – Wednesday 11/20

Learning Goals

Build **simulations** to study how systems change over time

- Intercept **events** and use them to modify simulation models
- Use **randomness** and **Monte Carlo methods** to derive truth from simulation

Event-Based Simulation

Time vs. Events

Last week, we discussed how to build a simulation by running rules regularly over time.

Today, we'll take a different approach: build a simulation by running rules **when an event occurs**.

Interaction Events

An **event** represents a single user interaction with the computer system. Events come in many forms: **keyboard presses**, **mouse clicks**, touchpad gestures, touchscreen presses, button presses, etc...

When you trigger an event on your computer, a **signal** is sent from the computer hardware to any programs that are currently running. That signal has information about the type of the event (key press vs. mouse click), plus any additional information that might be useful (which key was pressed).

Event Loop

Similar to the time loop that we used last time, we'll need to run an **event loop** to capture the signals that the computer sends out. However, events occur **irregularly,** unlike regularly-timed rules.

To implement this event loop, we'll have our simulation system constantly **listen** for events. When an event occurs, the simulation system will catch it, then send it on to a function we write specifically to handle that kind of event. This is done with a special kind of Tkinter function called **bind**, and is provided in the starter code.

Tkinter Events

With Tkinter, we can listen for and bind functions to lots of different event types.

We'll care about just two: **<Key>**, a keyboard press, and **<Button-1>**, a mouse click.

There are lots of other Tkinter events we can implement if we want them:

https://effbot.org/tkinterbook/tkinter-events-and-bindings.htm#events

Event Handlers

To deal with Key and Mouse events, we'll introduce two new functions to our simulation framework:

- `keyPressed(data, event)`
- `mousePressed(data, event)`

Each of these takes data (the components data structure), and an **event object**, which contains the information about the event.

These work like `runRules(data, call)` – we update data, then refresh the view right afterwards. This lets us make changes to the model!

keyPressed

In keyPressed, the **event** parameter contains two values we can use:

- `event.char` is a string containing the character pressed
- `event.keysym` is a string holding the 'name' of the character, for characters we can't represent as a string (like Enter or Backspace)

If we want to draw the last-pressed character in the middle of the screen, we'd need to store that character:

```
def keyPressed(data, event):
    data["text"] = event.char
```

mousePressed

In mousePressed, the **event** parameter holds pixel location where the user clicked on the canvas.

- `event.x` is the x location
- `event.y` is the y location

If we want to move a circle around the canvas to be centered wherever you click, we'd need to store the center location:

```
def mousePressed(data, event):
    data["cx"] = event.x
    data["cy"] = event.y
```

Advanced Example

We can use mouse and key events to improve our zombie simulation!

For mousePressed, change a human to a zombie if you click on them.

You can detect if you've clicked in a cell by calculating the cell's bounds, then checking if:

```
left <= event.x <= right
          AND
top <= event.y <= bottom
```

For keyPressed, restart the simulation if the user presses 'r'.

We can do this by checking if `event.char` equals 'r'.

Resetting the simulation is easy- just **reset the model** by calling `makeModel(data)`.

Using Mouse/Key Events

In general, we use events to let the user **directly change the simulation**.

This lets us observe how the simulation changes given a specific occurrence.

Randomness & Monte Carlo

Computing Randomness

We've already used Python's random library to produce random behavior. But we haven't discussed how this is possible.

Randomness is difficult to define, either philosophically or mathematically. Here's a practical definition: given a truly random sequence, there is **no gambling strategy possible** that allows a winner in the long run.

But computers are **deterministic-** given an input, a function should always return the same output. Circuits should not behave differently at different points in time. So how does the random library work?

True Randomness

To implement truly random behavior, we can't use an algorithm. Instead, we must gather data from **physical phenomena** that can't be predicted.

Common examples are atmospheric noise, radioactive decay, or thermal noise from a transistor.

This kind of data is impossible to predict, but it's also slow and expensive to measure.

Pseudo-Randomness

Most programs instead use **pseudo-random numbers** for casual purposes.

A **pseudo-random number generator** is an algorithm that produces numbers which look 'random enough'. These algorithms generally work by taking as input a number x_i, then running it through an algorithm to calculate x_{i+1}. By calling the function repeatedly on the numbers it outputs, we can generate a **sequence** of numbers.

Though these numbers aren't *truly* random, they're random enough that almost no one will be able to predict them. But we can make them predictable by **seeding** the algorithm to start from a specific number.

Python's Random API

Python's random library uses an algorithm called the Mersenne Twister to generate pseudo-random numbers

We've already discussed some functions:

- **random.randint(x, y), random.choice(L), random.shuffle(L)**

It also has some functions that let you directly access the generator:

- **random.seed(x), random.getstate()**

Many functions are based on **random.random()**, which generates a random floating point number in the range [0.0, 1.0)

Randomness in Simulation

Many simulations use randomness in some way; otherwise, every run of the simulation will produce the same result.

This means that the same simulation might have multiple different outcomes on the same input model. That makes it hard to estimate the true average outcome.

To find the truth in the randomness, we need to use probability!

Law of Large Numbers

The Law of Large Numbers states that if you perform an experiment multiple times, the average result will approach the **expected value** as the number of trials grows.

This works for simulation as well! We can calculate the expected value of an event by simulating it a large number of times.

We call programs that do this **Monte Carlo methods**, after the famous gambling district in the French Riviera.

Monte Carlo Method Structure

If we put our simulation code in the function runTrial(), a Monte Carlo method will typically take the following format:

```
def getExpectedValue(trials):
    count = 0
    for trial in range(trials):
        result = runTrial() # run a new simulation
        if result == True: # check the result
            count = count + 1
    return count / trials # return the probability
```

Monte Carlo Example

Every year, SCS holds the Random Distance Race. The length of this race is determined by rolling two dice. **What is the expected number of laps a runner will need to complete?**

```
import random
def runTrial():
    return random.randint(1, 6) + random.randint(1, 6)
def getExpectedValue(trials):
    lapCount = 0
    for trial in range(trials):
        lapCount += runTrial()
    return lapCount / trials
```

You do: what are the odds that a runner will need to run 10 or more laps?

Advanced Example

We can even use Monte Carlo methods on the zombie simulation we did last week!

Transfer the makeModel and runRules code to all take place in a single function (where the time loop becomes a while loop). Have that function return the number of days it takes to zombify all the humans.

When we run this function with getExpectedValues, we can find the expected amount of time left for the human race.

Learning Goals

Build **simulations** to study how systems change over time

- Intercept **events** and use them to modify simulation models
- Use **randomness** and **Monte Carlo methods** to derive truth from simulation