ENGR 40M Project 3c: Coding the raindrop pattern

For due dates, see the overview handout

The raindrop pattern works like this: Once per time period (say, 150 ms),

(a) "move" the pattern one plane down: the top plane to the second plane, the second to the third, and the third to the bottom; and

(b) choose some number of LEDs in the top plane at random to light up.

If you configure the time step and the number of new LEDs appropriately, this pattern looks like rain falling.

To make this pattern, we'll use the abstractions you wrote in lab 3b, so that we can focus on generating the `pattern[4][4][4]` array. Just as we did in lab 3b, we'll *decompose* this into smaller functions: a function `movePatternDown()` for step (a) and a function `chooseRandomInTopPlane()` for step (b).

We'll start by having just a single randomly-selected LED on the top plane light up, but this should be easy enough to extend to several LEDs.

This handout assumes you're working with a cube. If you have a plane, we leave it to you to modify the code to start on the top row and drip down, rather than the top plane.

1 Moving the pattern down

First, we'll write a function `movePatternDown(pattern)` that moves the pattern in the `pattern` array down by one level. That is, when `movePatternDown()` returns,

- the bottom plane should be the old second-from-bottom plane,
- the second-from-bottom plane should be the old second-from-top plane,
- the second-from-top plane should be the old top plane, and
- the top plane should be blank (all LEDs off).

There are lots of ways to do this, but you'll probably want a triply-nested for loop of some description (see slide 9 of lab lecture 3b for a reminder). Don't forget that you need to explicitly set every element in the top plane to turn the LED off.

> **L1:** Write `movePatternDown()` to implement the functionality described above.

```
void movePatternDown(byte pattern[4][4][4]) {
  // TODO: Write code to move the pattern down by one plane, and turn the
  // top plane off.

  // Here's a triply-nested for loop to get you started, but you can modify
  // or replace this if you wish:
  for (byte z = 0; z < 4; z++) {
    for (byte y = 0; y < 4; y++) {
      for (byte x = 0; x < 4; x++) {

      }
    }
  }
}
```

2 Choosing a random LED

The Arduino function `random(min, max)` generates a random number between `min` (inclusive) and `max` (exclusive). For example, `random(0, 3)` randomly chooses between `0`, `1` and `2` (but *not* 3). We want to write something that will pick a random (x, y) coordinate, since we know that we want the z-coordinate to be 3 (or whatever your top plane is). We'll need to pick two random numbers in the range of the cube size.

> **L2:** Fill in the function below to choose two random numbers and set them as your x and y coordinate (complete this even if you have a plane, and then modify your actual code on your laptop). After you've chosen the numbers, set the LED at that position in the top plane to turn on.

```
void chooseRandomInTopPlane(byte pattern[4][4][4]) {
    // Assumes the top plane has already been cleared (all lights set off)
    // TODO: Choose an x-coordinate and y-coordinate each at random
    //   [write your code here]

    // TODO: Set the value for this random (x,y) in the top plane (z=3)
    // of your pattern so that the LED will be on.
    //   [write your code here]

}
```

3 Moving the light down the cube

We wish to call the functions `movePatternDown()` and `chooseRandomInTopPlane()` regularly, once per time step (say, 150 ms). You've already used time-based programming in the useless box, but we'll review it here as well.

Arduino has two built-in functions to keep track of time: `delay()` and `millis()`. When `delay()` is executed, the program stops until the delay time has passed. Since the Arduino is essentially not running during this time, it won't be able to react to any user input during the `delay()`. The `millis()` function, on the other hand, just reads an internal clock (that runs independently of the software) and returns its value.

One strategy for updating every `TIME_DELAY` milliseconds goes like this: We maintain a `long` variable `nextUpdateTime` that stores the `millis()` clock value *at which the next update should take place*. If the current `millis()` exceeds `nextUpdateTime`, then we update the display, and bump `nextUpdateTime` to the *next* time at which an update should take place.

All of this code will go into your `loop()` method. This method runs continuously, so every time we go through, we can read `millis()` and see if enough time has elapsed.

There's one catch here: if we define a `nextUpdateTime` within `loop()` as a normal `long`[1], it will be a local variable, which means its value won't be stored when `loop()` runs again. To solve this problem, we define it as a `static long`. Static variables are initialized once, when the function first runs, and their value is stored even when the function is run again.

You can also use a global variable here, which is defined at the top of the program outside of any functions and can be accessed anywhere within your code. However, for good coding style, we generally try to avoid global variables so we can't accidentally change their value where we don't mean to. In general, variables should only be accessible by the functions that directly need them.

[1] We use long instead of int because `millis()` returns a larger value than int can store.

L3: Fill in the code below to check if enough time has elapsed, and if it is, print "update" to the serial port, call the two functions you wrote above, then update `nextUpdateTime`. Test this code to see if it works before continuing. It should look like a raindrop pattern!

```
const int TIME_DELAY = 150;

void loop() {
    static byte ledOn[4][4][4];
    static long nextUpdateTime = millis();

    // TODO: Write code to check if the current millis() clock is at least
    // nextUpdateTime. If it is, print "update" to the serial port, move the
    // pattern down, choose a (new) top plane, and update the nextUpdateTime
    // variable to the next time at which an update should occur.
    //   [write your code here]


    display(ledOn);
}
```

That's your basic raindrop pattern! Congraulations! Now, let's add some interactivity.

4 Adding user input

The raindrop pattern looks great, but what if you want to pause it to admire its stationary beauty? Here is some basic interactivity we might add, using the serial port:

- When the user types `'g'`, the raindrop pattern starts.
- When the user types `'p'`, the raindrop pattern freezes (pauses).
- When the user types `'s'`, the raindrop pattern stops, that is, turns all LEDs off.

We've already done this: you programmed a finite state machine that reacts to user input in the useless box. In this lab, we'll start with three states: "go", "pause" and "stop", corresponding to the states listed above.

L4: Draw a state machine for the three states of the cube. Your state machine should include the states and the conditions for transitioning between states, namely, the characters the user would type for the state to make the transition.

We're going to use a `switch` statement to figure out what state we're in and take the appropriate action. If you don't remember exactly how to code a `switch` statement, don't worry; we've written the basic outline, so you'll just need to fill it in.

> **L5:** Finish the `switch` statement below, and copy over the code you wrote in L3 above to an appropriate location in the code below, to execute your state machine.

```
void loop() {
    static byte pattern[4][4][4];
    static long updateTime = millis();
    static byte state;

    byte user_input = 0;
    if (Serial.available() > 0)
        user_input = Serial.read();

    switch(state) {
        case 'g':
            // TODO: Copy over your code to animate the raindrop pattern

            break;
        case 'p':
            // TODO: Fill in the 'pause' state to display the paused pattern

            break;
        case 's':
            // TODO: Fill in the 'stop' state to turn off everything

            break;
    }
}
```

Upload this program to your cube, and open the Serial Monitor to test it out. **Note: You will need to change to "No Line Ending" in the bottom right corner of the Serial monitor for this to work correctly.** Show your TA when you have it working.

5 Next steps

Now that you know how to create a simple pattern on the cube, try coding a few patterns of your own—either the ones you brainstormed in the prelab or a new one you've come up with since then. You can try one of the ideas below or come up with your own:

- Add variable brightness to the raindrop pattern so the LEDs fade in and out instead of just being on and off
- Set the cube to have multiple raindrops falling at a time
- Pulse the brightness of the cube from the center outwards (like a heartbeat)
- Code a snake-like pattern that will turn in a random direction whenever it hits the edge of the cube

> **L6:** Add at least one additional pattern to work on your cube. This should meet the complexity requirements outlined in the first section of this lab. You can use any type of user input you want—please consult the overview handout for more on different user input methods.

6 Reflection

Individually, answer the questions below. Two to four sentences for each question is sufficient. Answers that demonstrate little thought or effort will not receive credit!

R1: What was the most valuable thing you learned, and why?

R2: What skills or concepts are you still struggling with? What will you do to learn or practice these concepts?

R3: If this lab took longer than the regular 3-hour allotment: what part of the lab took you the most time? What could you do in the future to improve this?