Class 3

Working with Patches

For today's assignment, you will need to start with and build upon a working version of the Game of Life model, as programmed in the first part of *Wilenksy and Rand*, Chapter 2. If you have not created the model, or are uncertain of the correctness of your version, you may download a working version from Canvas. Please code the following procedures in the file and submit it on canvas. The NetLogo file you submit should be called *Class3.nlogo*.

3.1 Basic

1. Write a command procedure called `basic1` that when executed calls `clear-all` and then sets the color of all patches whose x-coordinate equals 6 or more to yellow.

   Suggestion: Create a button on the interface in order to more easily call and test `basic1`.

   Here's a stub for you to work with:

   ```
   to basic1
     clear-all
     [** Your code here. **]
   end
   ```

   Answer:

   ```
   to basic1
   ```

```
  clear-all
  ask patches with [pxcor > 5] [set pcolor yellow]
end
```

---

2. Write a command procedure called `basic2` that when executed calls `clear-all` and then colors each patch yellow with probability 12/100, and then for each patch if it has 4 or more yellow neighbors, sets the patch color to red and sets it label to `dead`.

   Here's a stub for you to work with:

```
to basic2
  clear-all
  [** Your code here. **]
end
```

Answer:

```
to basic2
  clear-all
  ask patches [if random 100 < 12
    [set pcolor yellow]]
  ask patches [if count neighbors with [pcolor = yellow] >= 4
    [set plabel "dead"
      set pcolor red]
  ]
end
```

---

3. Make a new slider called `spawn-percentage` on the interface. It should range from 0 to 100 in increments of 1. Change the `setup` method so that, on average, `spawn-percent`%, instead of always 10%, of the patches begin alive at setup.

   Consider: Test what are the lowest and highest percentages at which life can survive beyond the first few ticks.

   You should modify the current setup procedure:

```
to setup
  ca
  ask patches [
```

```
    set pcolor blue - 3
    if random 100 < 10 [ set pcolor green ]
  ]
  reset-ticks
end
```

Answer:

```
to setup
  ca
  ask patches [
    set pcolor blue - 3
    if random 100 < spawn-percentage [ set pcolor green ]
  ]
  reset-ticks
end
```

---

4. Write a new setup procedure called `make-glider` that clears the world and makes a glider with its top-left corner at (0,0). The glider should move down and to the right, as shown in the book (see pg. 64 of *Wilensky and Rand*).

   Here's a stub for you to work with:

```
to make-glider
  clear-all
  [** Your code here. **]
end
```

Answer:

```
to make-glider
  ca
  ask patches [
    set pcolor blue - 3]
    ask patch 0 0 [set pcolor green]
    ask patch 1 -1 [set pcolor green]
    ask patch 2 -1 [set pcolor green]
    ask patch 0 -2 [set pcolor green]
    ask patch 1 -2 [set pcolor green]
```

```
  ]
  reset-ticks
end
```

---

5. Find a still-life configuration with exactly five live cells and write a new procedure called `make-still` which clears the world and creates the still configuration near the center of the view. When `make-still` is run, there should be exactly five green cells on the view, and nothing should change when `go` is run. See pg. 60-63 of *Wilensky and Rand* for more detail on what a still-life configuration is.

   Hint: If you get stuck, try googling game of life still life configurations.

   Here's a stub for you to work with:

```
to make-still
  clear-all
  [** Your code here. **]
end
```

Answer:

```
to make-still
  ca
  ask patches [
    set pcolor blue - 3]
    ask patch 0 0 [set pcolor green]
    ask patch 0 -1 [set pcolor green]
    ask patch 1 0 [set pcolor green]
    ask patch 2 -1 [set pcolor green]
    ask patch 1 -2 [set pcolor green]
end
```

---

6. Find a period two oscillator different from the blinker and write a new procedure called `make-oscillator` which creates it near the center of the view. See pg. 63 of *Wilensky and Rand* for more detail on what an oscillator is.

   Hint: If you get stuck, try googling game of life oscillators.

   Here's a stub for you to work with:

```
to make-oscillator
  clear-all
  [** Your code here. **]
end
```

Answer:

```
to make-oscillator
  ca ask patches [
    set pcolor blue - 3
    ask patch 0 0 [set pcolor green]
    ask patch 1 0 [set pcolor green]
    ask patch 2 0 [set pcolor green]
    ask patch 1 -1 [set pcolor green]
    ask patch 2 -1 [set pcolor green]
    ask patch 3 -1 [set pcolor green]
  ]
  reset-ticks
end
```

3.2 Beyond the Basics

1. Write a procedure called `go2` which implements an altered set of transition rules. You should begin with a copy of the go procedure, but make the following changes. If a patch is set to die in the old rules (ie. change to `blue - 3`), instead of changing its pcolor to `blue - 3`, change it to `yellow`. All yellow patches should die (ie. change to `blue - 3`) each tick, before anything else happens. Because of this, neighboring yellow patches should not be counted when deciding whether a cell should die or come to life. After yellow cells are changed to blue, the procedure should proceed as normal. Note that it is possible for a patch to first change from yellow to blue, then back to green again, in the same tick. When coded correctly, this procedure will have a kind of after-image effect, in which cells fade to yellow before dying completely.

   Suggestion: Create a forever button to call `go2`, in order to more easily test it.

You should create a copy of the `go` procedure named `go2` and make the alterations in it. Don't forget to call tick at the end of the procedure.

Answer:

```
to go2
  ask patches [
    set live-neighbors count neighbors with [pcolor = green]
  ]
  ask patches [
    if pcolor = yellow [set pcolor blue - 3]
    if live-neighbors = 3 [set pcolor green]
    if (live-neighbors = 0 or live-neighbors = 1)
        and pcolor = green [set pcolor yellow]
    if live-neighbors >= 4 and pcolor = green [set pcolor yellow]
  ]
  tick
end
```

---

2. Create four sliders, `min-spawn`, `max-spawn`, `min-over-pop`, and `max-under-pop`, each of which scales between 0 and 8 in increments of 1. Write a new procedure called `go3` on the model of `go`, except replacing the hard-coded numbers with slider values. If the number of a cell's live neighbors is both greater-than-or-equal-to `min-spawn` and less-than-or-equal-to `max-spawn`, the cell should come to life (ie. turn green). In other words, `min-spawn` and `max-spawn` are the lower and upper bounds, respectively, on neighbors for the birth of new life. On the other hand, if the number of a cell's live neighbors is either less-than-or-equal-to `max-under-pop` or greater-than-or-equal-to `min-over-pop`, the cell should die (ie. change color to `blue - 3`). In other words, `max-under-pop` is the maximum number of neighbors for which a cell will die due to underpopulation, and `min-over-pop` the minimum number of neighbors for which a cell will die due to overpopulation. Putting in slider variables in place of the hard-coded numbers generalizes the rules of the game, allowing you to run different kinds of rule configurations. Note that the rules of the original game are as follows: `min-spawn = 3`, `max-spawn = 3`, `max-under-pop = 1`, `min-over-pop = 4` (Think about why this is the case).

Consider: Do all configurations of the four sliders make sense as valid rule sets? Consider, for instance, the setting in which `max-under-pop` is greater than `min-spawn`. In these configurations, it is unclear what the procedure should do. Note that so long as your procedure handles the valid settings, you will pass the automated tests. Of the configurations that do make sense, which ones make for interesting games, and which result in all cells either dying out or coming to life?

You should create a copy of the `go` procedure named `go2` and make the alterations in it. Don't forget to call tick at the end of the procedure.

Answer:

```
to go3
  ask patches [
    set live-neighbors count neighbors with [pcolor = green]
  ]
  ask patches [
    if live-neighbors >= min-spawn
        and live-neighbors <= max-spawn [set pcolor green]
    if live-neighbors <= max-under-pop [set pcolor blue - 3]
    if live-neighbors >= min-over-pop [set pcolor blue - 3]
  ]
  tick
end
```

Class 4

Working with Turtles

For today's assignment, you will need to start with and build upon a working version of the Game of Life model, as programmed in the first part of *Wilenksy and Rand*, Chapter 2. If you have not created the model, or are uncertain of the correctness of your version, you may download a working version from Canvas. Please code the following procedures in the file and submit it on canvas. The NetLogo file you submit should be called *Class3.nlogo*.

4.1 Basic

1. First create a slider on the interface called `num-turtles` which ranges from 1 to 100 in increments of 1. Then write a command procedure called `basic1` that when executed calls `clear-all` and then creates `num-turtles` green turtles at random locations on the view. Set each turtle to be facing to the right and color the patch under it blue (Hint: you should be able to do all of this in the block you use to create the turtles).

   Suggestion: Create a button on the interface in order to more easily call and test `basic1`.

   Here's a stub for you to work with:

   ```
   to basic1
     clear-all
     [** Your code here. **]
   end
   ```

Answer:

```
to basic1
  ca
  crt 15 [
    setxy random-xcor random-ycor
    set color green
    set heading 90
    set pcolor blue
  ]
end
```

---

2. Write a command procedure called `basic2` that when executed calls `clear-all` and then creates `num-turtles`. All turtles should be orange and begin in random coordinates with random headings.

   Here's a stub for you to work with:

```
to basic1
  clear-all
  [** Your code here. **]
end
```

Answer:

```
to basic1
  clear-all
  ask patches with [pxcor > 5] [set pcolor yellow]
end
```

---

3. Find a new non-frozen configuration of the Heroes and Cowards model (ie. the turtles should never stop moving and freeze into a static configuration. For more information, see *Wilensky and Rand* pg. 78). Write a command procedure called `non-static` that when executed calls `clear-all` and then

   Hint: You can do this in one line of code, if you use the pre-existing code in the model. Look at the `preset [seed]` procedure in the Heroes and Cowards model you started with, as well as the built-in preset starting configuration buttons.

   Here's a stub for you to work with:

```
to basic1
  clear-all
  [** Your code here. **]
end
```

Answer:

```
to non-static
  preset 110
end
```

---

4. Write a command procedure called `basic3`

   Here's a stub for you to work with:

```
to basic1
  clear-all
  [** Your code here. **]
end
```

Answer:

```
to basic1
  clear-all
  ask patches with [pxcor > 5] [set pcolor yellow]
end
```

4.2 Beyond the Basics

1. The Heroes and Cowards model is non-deterministic, that is, given the same initial configuration, it will run differently every time it is run. This non-determinism arises from some amount of randomness in the `go` procedure. However, this amount of randomness is very small, thus different runs are nearly identical. Looking at the `go` procedure, determine what about it is non-deterministic. Then write a new procedure called `go-deterministic` which exhibits almost identical, but deterministic behavior.

   Hint: Think about the `ask turtles` block when figuring out where the randomness is coming from. There are a few different ways you

could do change the implementation. Try do choose the one that results in minimal changes to the code.

Consider: Does this change affect the exhibited behavior of the model? Think about why or why not it might. (Look at the yo-yo preset and compare behavior.)

You should make a copy of the existing `go` procedure called `go-deterministic` and then modify it to have to correct behavior.

Answer:

```
to go-deterministic
  ask turtles [
    if (color = blue)  [
      facexy ([xcor] of friend + [xcor] of enemy) / 2
          ([ycor] of friend + [ycor] of enemy) / 2
    ]
    if (color = red)   [
      facexy [xcor] of friend +
          ([xcor] of friend - [xcor] of enemy) / 2
          [ycor] of friend +
          ([ycor] of friend - [ycor] of enemy) / 2
    ]
  ]
  ask turtles [
    fd 0.1
  ]
  tick
end
```

---

2. Write two procedures, `tag-setup` and `tag-go` which encode the rules for the game of tag. That is, begin with `num-players` players, one of which, chosen at random, is red and designated "it". All the rest of the turtles should be blue. Each turn, the "it" should select the closest turtle to it and move 1.0 towards that turtle. All other turtles should move randomly (ie. select a random heading each turn and move forward 1.0. When the "it" and a blue turtle are in the same patch during the same turn, before moving, they switch colors, sot that the blue turtle becomes "it" and the old "it" turtle becomes blue.

Answer: