Interacting Patches

In the previous two labs we have used turtles as mobile agents that move according to set rules and form spatial patterns. It is often the case that patterns and order emerges as a result of interactions that take place between stationary agents. Examples include growth of forests, the growth of algae, or the interactions between cells in an organism. In this workshop we will use NetLogo to model such interactions using patches. We will see that although the patches themselves do not move, waves of changes within the patches can mimic various types of complex movement. It is believed that such interactions are responsible for the complex patterns we see in animal coats, and perhaps at a more fundamental level for cell differentiation of a developing embryo.

The Game of Life

The mathematical concept behind the model we will build is that of a cellular automata. The fundamental idea is that each patch can have a number of different states and it changes states by examining the states of its neighbors and following some simple rules. We will start with the well-known example of Conway's Game of Life, and build from this to create a somewhat realistic model of the growth of stationary species. The model can be extended readily to include the sorts of chemical reactions that play a role in animal development.

In the Game of Life, patches can have two states: alive and dead. The patches change their states according to the following rules: A patch that is dead will come to life in the next generation if it has three live neighbors. A patch that is alive will only continue to live if it has either two or three live neighbors. In this model we will let a patch that is alive have the state 1 and a patch that is dead have the state 0.

Start your model by defining the state variable for the patches:

```
patches-own [state]
```

Then write a procedure called `random-setup` that randomly assigns either a 0 or a 1 to the state variable of each patch and then colors patches that have state 1 white and those with state 0 black. An efficient way of doing this is

```
to random-setup
      clear-all
      ask patches [
            set state random 2
            set pcolor scale-color red state 0 1 ]
end
```

The line in which the `pcolor` is defined may seem funny (why are we using red when we want white and black?). `scale-color` returns a color value corresponding to a shade of the color specified, but black when the state variable is 0 and white when the state variable is 1. If the state variable was between 0 and 1 it would give a color which is a shade of red. There are certainly other ways of doing this, but this one will generalize naturally to the extensions you will do later.

Rules of the Game
Now it is time to apply the cellular automata rules for the Game of Life

We will do this in a `go` procedure. The only thing this procedure needs to do is change the state of a patch according to the rules of the Game of Life. However, there is one subtlety you should be aware of. Since NetLogo executes commands to different patches sequentially if you are not careful one patch may change its state, before its neighbor has had a chance to ask it what state it was in. In order to keep all changes in step, we will first ask each patch to keep a record of the total number of its neighbors that are alive (state 1). Only once all of the patches have done this will we allow them to change state. The rule to change the state can be applied efficiently by using the an `ifelse` statement which allows you to execute different commands when different conditions apply. Nested `if` statements are used when more than two conditions apply

```
to go
      ask patches [set total count neighbors with [state = 1]]
      ask patches [
            ifelse total = 3
                  [set state 1]
                  [if total != 2 [set state 0] ]
            set pcolor scale-color red state 0 1 ]
end
```

In order for the above code to be complete you need to add the variable `total` to the `patches-own` list. Notice the use of the `neighbors` primitive. This returns a list of the 8 patches surrounding the patch in question. Make sure you follow the logic of the if statement in this procedure. Later you may want to change the rules to create your own cellular automata. Create a button for the `go` procedure and test it.

Designing Life
After some initial craziness you should see the patches settle down to a few simple stable structures and perhaps a few blinking patterns. It may not appear too “life”-like but in fact there are a whole host of interesting “organisms” in the Game of Life, some of which appear to move (gliders), others which breed (breeders). If you run your game a few times you might see a few of these emerge. To see the more complicated “organisms” you need build them. There is no blind watchmaker in the Game of Life. Although it has been shown that it is theoretically possible to create self replicating “molecules” á la DNA, these are not stable to invasion by “parasites” and have no hope of evolving. However, we can be “watchmakers” by drawing “organisms” on a blank canvas. We can do this by using NetLogo's ability to track the location of the mouse.

First create a `clean-slate` procedure that sets all the patches to be dead

```
to clean-slate
      clear-all
      ask patches [
            set state 0
            set pcolor black ]
end
```

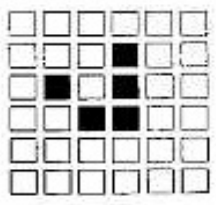
Now create a new procedure that will turn the patch at the location of the mouse to be alive

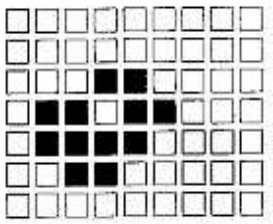
(white and state 1) when the mouse is pressed.

```
to draw
       if mouse-down? [
              ask patch mouse-xcor mouse-ycor [
                     set state 1
                     set pcolor white ] ]
end
```

Since it is likely you will also want to erase mistakes, create a similar procedure called `erase` that you can use to "kill" a patch. The `draw` and `erase` procedures should be executed with "run forever" buttons. You may find it useful to put a command in your `go` procedure to stop if the mouse is down.

Play around with this code for a while and see what life forms you can create. Try drawing the two patterns below.





At this time you may want to try modifying the rules of the game in some way to see what other "lifescapes" you can create.

Assignment: Extending the Model

Although the "organisms" you created in the Game of Life may mimic biology, they are not a biological model per se. However, we can easily modify the code to model different situations in biology and chemistry. Here is what I propose, Let's allow the `state` variable in the above program to be the age of some organism. (I suggest you change the name to `age` to make this explicit in your code). We will have patches get older each year until they reach a maximum age, at which point they are considered dead (and we set their age to 0). Patches will only die by reaching old age. Let's have a model where new cells can only be born in patches that are dead and which also have the exactly the right number of fertile neighbors. We could call these fertile neighbors parents.

1. First rescale your screen to be 50 by 50 with patch size 5
2. Let the `age` variable range from 0 to `max-age`. Were `max-age` can be specified by a slider (choose a slider range for `max-age` to be 1 to 100).
3. Since we have cells ranging in ages from 0 to `max-age`, modify your `random-setup` procedure to create patches of all ages. Also make sure you change your `scale-color` range to include the full spectrum of ages.
4. Now modify the rules. Let a patch be born in a dead cell if the dead cell has two "parents". An organism can be a parent when it has reached a fertile age and is not older than a sterile age. For now let the age of fertility be 15 and the age of sterility be 50.
   (a) Change your `go` procedure so that `total` is the number of neighboring patches who could be parents.
   (b) Modify the next part of your `go` procedure to ask the patches which are not dead to get older by one year and then if their age is greater than the maximum age to die. For

those that are already dead (age 0) ask them to be reborn if they have exactly 2 parents.

(c) Make sure your command to color the patches is scaled correctly.

5. Run your modified program. Experiment with different values for `max-age` to see how having non-reproductive neighbors affects the population distribution.
6. Modify your program so that you have sliders ranging from 0 to 100 for both the `fertile-age` and the `sterile-age`. Also add a slider from 0 to 8 for `parent-number`. Modify the appropriate lines in your program.
7. Add commands that will allow you to plot the total number of patches in each age group: immature, fertile, and sterile.
8. Now explore your model in detail. Experiment by starting with a random initial distribution of ages and with a clean-slate on which you seed your own pattern of newborn patches. You will find that some behavior can only be obtained by carefully tuning the sliders as the model runs.
9. With `max-age` set at 100 and `parent-number` set at 2 find parameter values for `fertile-age` and the `sterile-age` for which each of the following types of patterns happen. Write your answers on the information tab of your program.
   (a) The population exists only in isolated groups that appear to drift across the screen.
   (b) The changing population distribution appears to move in large lacey waves across the screen.
   (c) The changing population distribution resembles a tight spiraling pattern.
   (d) The population appears to change in a random way.
   (e) The majority of patches appear to change together in a synchronized way.
10. Try to get qualitatively different patterns using different values for `max-age` and `parent-number`. In particular, try getting “Life” like patterns with low values for `max-age`. Also try setting `parent-number` to 0. You might also want to consider modifying your code to allow for a range of acceptable neighboring parents for rebirth to take place (eg 2 or more parents would be a suitable model for some fruit trees).
11. Submit your completed program to our moodle site using the naming convention Lastname_Firstname_Lab4.nlogo