NetLogo Tutorial Series:
Mystery Pattern

Nicholas Bennett
nickbenn@g-r-c.com

August 2015

Copyright and license

Acknowledgments

Development of this curricular material and the accompanying NetLogo models was funded in part by:

- Santa Fe Institute Summer Internship/Mentorship (SIM) and Summer Complexity and Modeling Program (CaMP) for high school students;
- New Mexico Supercomputing Challenge;
- Project GUTS;
- New Mexico Computer Science For All (CS4All).

Participants in the above programs have also provided invaluable feedback on earlier versions of this material.

Introduction

Sometimes we find that simple behaviors, when repeated by many agents – or even by a small number of agents, repeating those behaviors many times – can produce rich, unexpected patterns.

In this activity, we'll be building a NetLogo model that has agents with a very simple behavior. In each step, an agent (we'll call it a "painter") will perform the following actions:

1. Select one of three target points (the vertices of a triangle) at random.
2. Move half the distance toward the selected target point.
3. Mark its new location with a white dot (on a black background).

To begin, each painter will be located on one of the three vertices. (Obviously, if a turtle selects the same vertex as its first target, it won't move in that iteration.)

What pattern (if any) will result from the painters following their programmed behaviors? How many repetitions of the above steps will it take before any such pattern emerges?

We could try to answer the questions above by doing the exercise on paper, but it would probably become very tedious, very quickly – possible long before the "mystery pattern" is clear.

Task 1: Getting Started

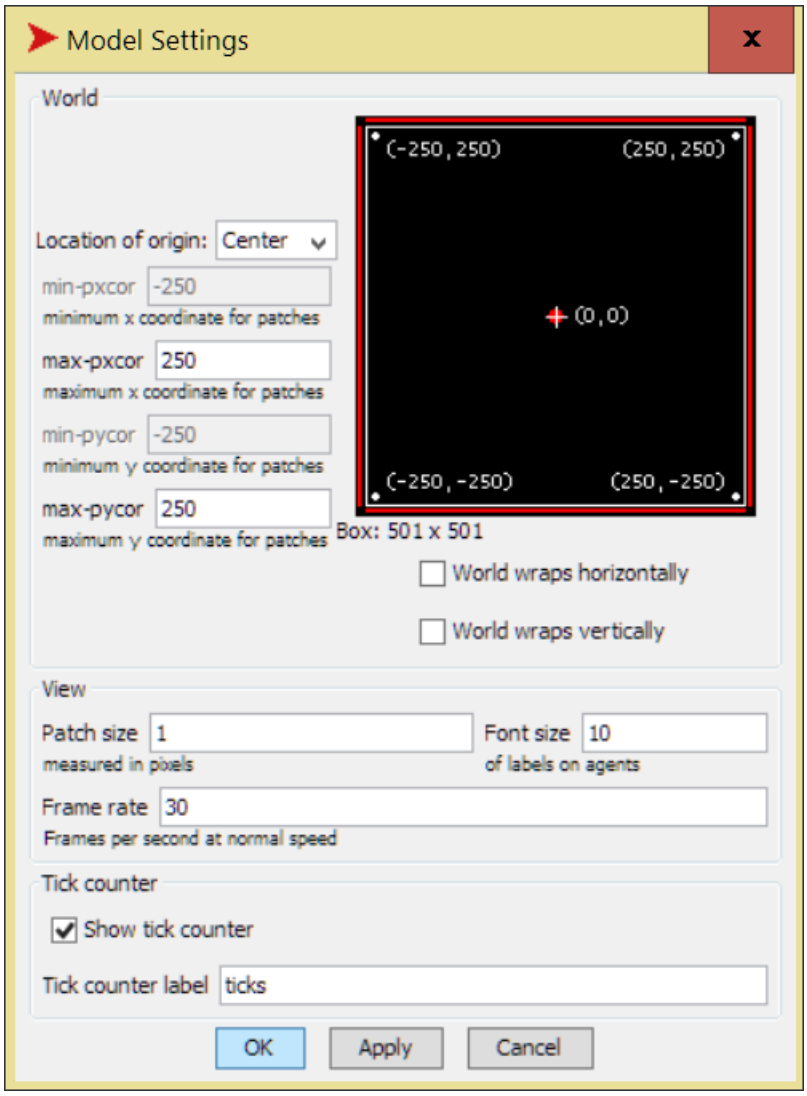
(Before following the steps below, consider reading "The NetLogo Coordinate System" in "NetLogo Tutorial Series: Introduction and Core Concepts". Even if you've read it already, it might be useful to review it.)

Start NetLogo & configure world

1. From the Macintosh **Applications** folder, or from the Windows **Start/All Program/NetLogo** menu, launch the **NetLogo**[1] application (not **NetLogo 3D**).
2. Because we want the marks made by the painters to be placed very precisely, and because NetLogo colors an entire patch at once, we want to set up the NetLogo world with a large number of very small patches. To do this, click the **Settings…** button at the top of the NetLogo **Interface** window, and make these changes:
   a) Leave **Location of origin** set to **center**.
   b) Set the **`max-pxcor`** value to 250.
   c) Set the **`max-pycor`** value to 250.
   d) Uncheck the **World wraps horizontally** checkbox.
   e) Uncheck the **World wraps vertically** checkbox.
   f) Set the **Patch size** value to 1.0.

1 This tutorial was originally written for NetLogo v4.x, then updated for and tested with NetLogo v5.0-v5.2. Though we recommend completing this tutorial with NetLogo v5.x, it should work with any 4.x or 5.x version. There are some slight differences in user interface and terminology between the versions, however. For example, there are some checkboxes and other inputs in the **Model Settings** and **Buttons** dialogs of NetLogo v5.x that aren't present in the NetLogo v4.x dialogs; these can be safely ignored when using this tutorial with NetLogo v4.x.

The **Model Settings** window should look like this:



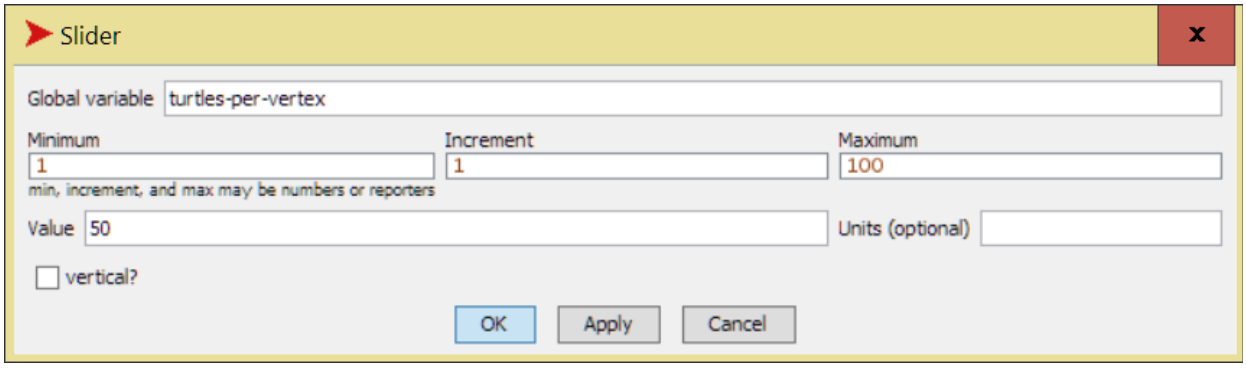
3. Click the **OK** button; we now have a world that's 501 patches tall and 501 patches wide, with each patch being 1 pixel X 1 pixel in size.

Controlling the number of painters

One experiment we'll do is to see if the results vary with the number of painters. Since all the painters will begin on one of the three vertices of a triangle, the easiest approach will be to control the number of painters per vertex; we'll use a slider to do this.

1. Select the **Slider** tool from the pull-down menu in the toolbar at the top of the screen.
2. Click somewhere in the white space to the left of the world; this will display the **Slider** configuration window.
3. Make the following changes:
   a) Set the **Global variable** value to **`painters-per-vertex`**.
   b) Set the **Minimum** to 1.
   c) Set the **Increment** to 1.
   d) Set the **Maximum** to 100.

   The Slider configuration window should now look something like this:



4. Click the **OK** button.

Shortly, we'll write the code that will use the value from this slider to create the specified number of painters – and then ask them to follow the behavioral rules described previously.

Task 2: Writing the model

Setting up the vertices of the triangle

Imagine drawing an equilateral triangle on the NetLogo world, with the top vertex centered almost at the top of the world, and the other two vertices the same distance from the origin as the top vertex, but located in the lower part of the world, to the left and right of center.

Take a few moments to consider how we might go about identifying these three vertices – more importantly, how we might have our painter agents use these points as targets.

As it turns out, it's much easier to send agents from the origin to these three points than it is compute their coordinates.

When we create turtles using the **`create-turtles`** command, the turtles are created at the origin, and each has a random heading and a random color, by default. But a less frequently used variant of this command, **`create-ordered-turtles`** (and the breed-specific version, **`create-ordered-`*`breeds`***) arranges the headings of the turtles created around the compass. For example, **`create-ordered-turtles 4`** will create 4 turtles, setting the heading of the first turtle to 0°, the heading of the second to 90°, the third to 180°, and the fourth to 270°.

Can you think of how we might use this behavior to set up our vertices?

Let's start writing our NetLogo code.

1. Click on the **Code** tab, near the top of the NetLogo window.
2. Type the following at the top of the code editing area, to tell NetLogo that we'll be using a breed called **`vertices`** to establish the vertices of the triangle:

   ```
   breed [vertices vertex]
   ```

3. Notice that a breed declaration consists of the **`breed`** keyword, followed by a set of square brackets. Inside the brackets are two symbols: the first is the plural name of the breed (also the name of an agentset containing all turtles of the specified breed); the second is the singular name of the breed.

4. Below what you just typed (we recommend giving yourself a blank line or two first), type the following **setup** procedure (remember, spaces, spelling, and square brackets are *very* important; parentheses are less critical, except where noted):

```
to setup
  clear-all
  create-ordered-vertices 3 [
    set color white
    forward (max-pycor - 5)
  ]
  reset-ticks
end
```

   As is the case for **create-turtles** and **create-*breeds***, when the number of turtles to be created by **create-ordered-turtles** or **create-ordered-*breeds*** is followed by a set of square brackets, each of the newly created turtles will execute the commands enclosed in the brackets. In this case, each **vertices** agents will change its color to **white**, and then move a distance of **(max-pycor - 5)** from the origin.

5. Click on the **Check** button near the top of the **Code** window. If any syntax errors are reported, try to fix them: Read the error messages carefully, noting the location highlighted in the code. and try to correct the issues reported. Hint: The most common syntax errors, by far, are caused by misspelling (or inconsistent spelling), accidentally substitution of spaces or underscores for the hyphen character (hyphens and underscores are not interchangeable), and failure to include necessary white space (note that there are spaces before the number 3, before the minus sign, and before the number 5; these are all required).

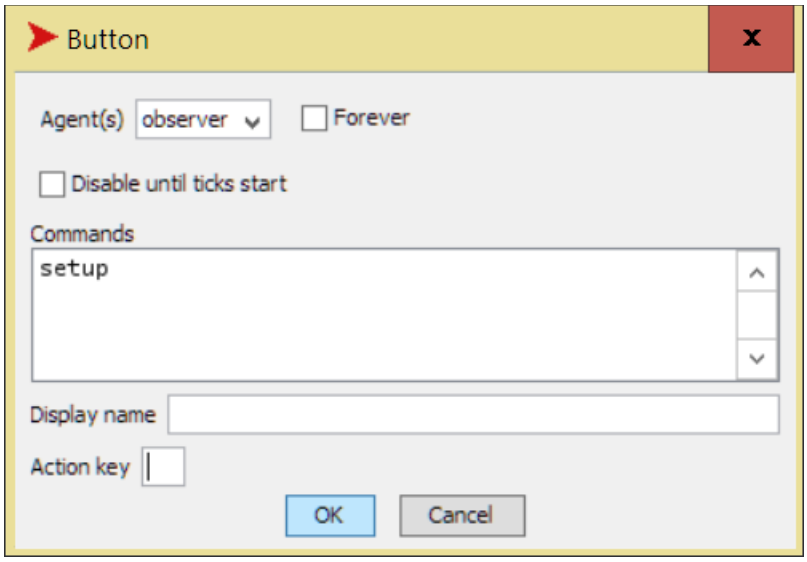
6. Review your code (whether any fixes are required or not). Can you predict what will happen with the **setup** procedure is run?

7. After any syntax errors are corrected, save your model.

8. Click on the **Interface** tab to display the window containing the NetLogo world and the user interface for your model.

9. Create a button to run the **setup** procedure you just wrote. (You can use the same pull-down menu near the top of the **Interface** window, or you can simply right-click in the white space, and select **Button** from the menu that pops up.)

10. Fill in as many of the fields in the Button dialog as you can, before checking your work against the image on the next page. Remember: Usually, a model's **setup** procedure must be run by the observer; this model is no different in that respect. Also, remember that providing values for **Display name** and **Action key** is optional.



11. Click **OK** to close the **Button** dialog.

12. Run your **`setup`** procedure by clicking the **setup** button. Are the results what you expected? (Hint: You should see three very faint white spots in the NetLogo world: one horizontally centered, a few pixels from the top; two more, one each on the left and right, both in the lower half of the world.

Creating the painters

Our next task is to declare another breed of turtle – painters, this time – and create some number of them (the number specified by the **`painters-per-vertex`** slider, in fact) on each of the vertices.

1. Add another **`breed`** declaration at the top of your code in the code window. It can be before or after the existing **`breed`** declaration, but it must be before any procedures.

   The plural name for this breed should be **`painters`**, and the singular name should be **`painter`**. (Note that the plural and singular names cannot be the same.)

   After you've added the declaration, check your work against the code shown on the next page. (Code that we wrote in a previous step is shown in gray italicized type, while code added in the current step is shown in the same colors used by the NetLogo editor.)

```
breed [vertices vertex]
breed [painters painter]

to setup
  clear-all
  create-ordered-vertices 3 [
    set color white
    forward (max-pycor - 5)
  ]
  reset-ticks
end
```

2. Take a few minutes to think about the best way to place our painters on the vertices. Should we use one or more **create-ordered-painters** commands, and have the painters follow the same paths as the vertices? Should we find out what patches the vertices agents are standing on, and have our painters move directly there? Maybe there's a simpler way than either of these possibilities.

   It might not be immediately and intuitively obvious, but a turtle of one breed can hatch a turtle of another breed. Normally, when one turtle hatches another, the hatchling is essentially a clone of the first: same size, same color, same location, same heading, same breed, etc. Of course, if one turtle hatches another of a different breed, there will be a more obvious difference between the two – but they'll still have much in common. In particular, they'll be in the same location.

3. Let's have our vertices agents hatch painters as soon as they are in position. To do this, we'll need to modify our **setup** procedure. (Once again, you can use the type style and color to distinguish between code we wrote previously – and mustn't write again – and code we're adding now.)

   Particularly if you have experience in writing code that uses **hatch** or **hatch-*breeds***, see if you can figure out what code to add to the **setup** procedure (and where), before referring to the code shown below.

```
to setup
  clear-all
  create-ordered-vertices 3 [
    set color white
    forward (max-pycor – 5)
    hatch-painters painters-per-vertex [
      set size 4
    ]
  ]
  reset-ticks
end
```

(Changing the size of the painters is entirely unnecessary, in terms of the model results – but it does make it easier to see the painters.)

4. Check and save your code.
5. Use the **setup** button to run the **`setup`** procedure again. Do you notice any difference?

Implementing the painters' behavior in the `go` procedure

The behavior of the painters in this model is quite simple – and even though we have two breeds of turtles, the job of the vertices agents, once the **`setup`** procedure is done, is just to stand in place. So we'll do something we won't do in most models: We'll write all the remaining code in the **`go`** procedure, rather than (for example) creating a **`paint`** procedure, and invoking that from the **`go`** procedure.

1. Before writing the **`go`** procedure, review the NetLogo Dictionary entries for the following primitives:
   - **`ask`**
   - **`distance`**
   - **`face`**
   - **`let`**
   - **`one-of`**
2. Review the behavior described in "Introduction" (p. 3), and consider how you might use the primitives listed above (along with any others you're familiar with, that you think might be useful) to implement the painters' behavior.
3. Create a **`go`** procedure to implement the painters' behavior. We suggest starting with the primitive(s) common to almost all **`go`** procedures (e.g. **`tick`**), then – keeping in mind that the observer will be executing the procedure – include code for asking all painters to follow the necessary steps.

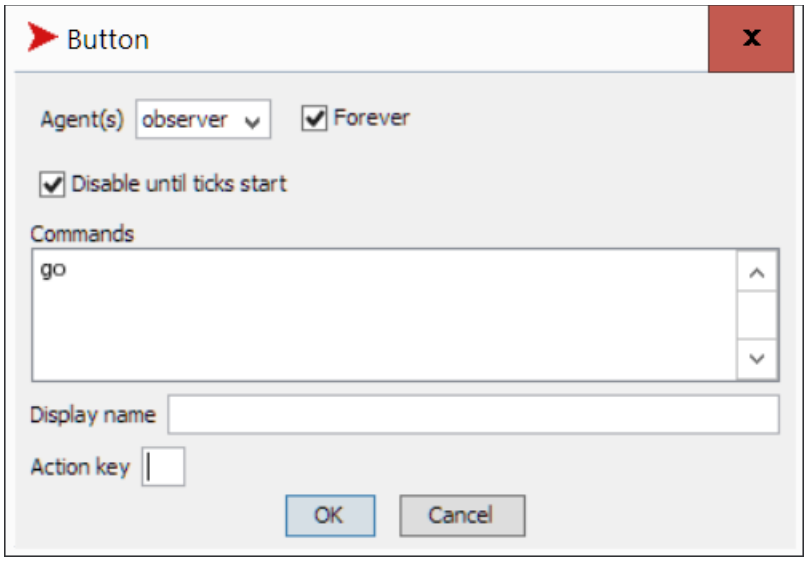
   See how much of the procedure you can write, before checking your work against the code shown on the next page. (This time, all of the program code is shown – however, the code written in previous steps is once again shown in gray italicized type.)

```
breed [vertices vertex]
breed [painters painter]

to setup
  clear-all
  create-ordered-vertices 3 [
    set color white
    forward (max-pycor – 5)
    hatch-painters painters-per-vertex [
      set size 4
    ]
  ]
  reset-ticks
end

to go
  ask painters [
    let target one-of vertices
    face target
    forward 0.5 * distance target
    set pcolor white
  ]
  tick
end
```

The key elements of this procedure are these:

- **`let target one-of vertices`** selects one of the **`vertices`** breed at random, and assigns it to the **`target`** local variable. (Remember that the plural name of a turtle breed is also the name of an agentset containing all turtles of that breed.)
- **`face target`** changes the heading of the painter so that it faces the selected vertex.
- **`forward 0.5 * distance target`** moves the painter forward by half the distance to the selected vertex.
- **`set pcolor white`** changes the patch color where the painter is now standingto white.

4. Check and save your code (fixing as necessary).
5. In the **Interface** window, add a button to invoke the **`go`** procedure. Keep in mind the agent that you think should run the button, and whether the button should run a single time, or repeatedly. Before clicking **OK**, compare your settings to those on the next page.



6. Click **OK**.
7. Once again, make sure your model is saved before testing it.

Task 3: Running the program

The moment of truth

Assuming you've managed to catch and correct any syntax errors along the way, your program should be ready to run. Have you formulated a hypothesis about the type of pattern – if any – that will result, when the painters start moving around and turning the patches they land on white? What do you think will happen?

Will it make a difference if you start with one painter per vertex vs. 100? Do the painters interact directly in any way? Do they interact indirectly – e.g. by contending for space or other constrained resources?

Let's try it:

1. Set the **`painters-per-vertex`** slider to a value of 1.
2. Click the s**etup** button.
3. Click the **go** button. If you get any errors, try and fix them – by yourself, or with the instructor's help.
4. Assuming your program is running correctly, what pattern do you notice?
5. Change the value of **`painters-per-vertex`** to a higher value, and repeat steps 2 and 3. Has anything changed?

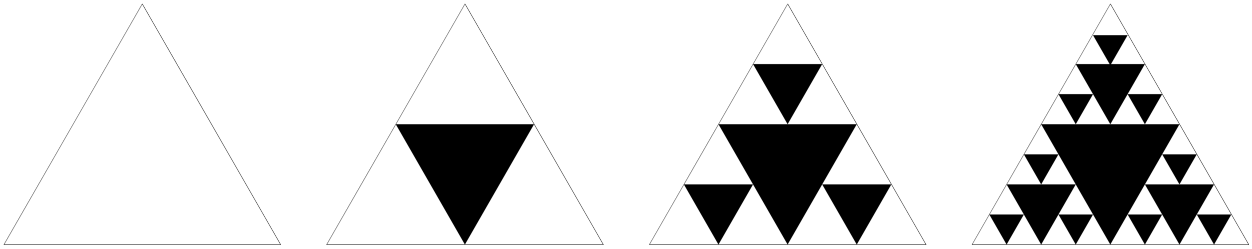
6. Repeat step 2, then slow your model movement down, using the slider at the top of the NetLogo world. Now repeat step 3, and watch how the painters move. How would you describe this movement, in general terms?

What's the mystery pattern?

The pattern drawn by the painters is called the Sierpinski triangle, or the Sierpinski gasket. The best-known recipe for creating this figure is as follows:

1. Start with a triangle.
2. From the original triangle, cut out the triangle formed by connecting the midpoints of all three sides of the original triangle.
3. Continue the process with the smaller triangles created, treating each one as if it were the original triangle.

The first few iterations of this process are illustrated below:



Actually, there are several methods for producing the Sierpinski gasket; the approach we used is a specific version of an algorithm called the *chaos game* [2]. In contrast with the construction method described immediately above, the rules followed in the approach we used give very few clues as to the final result – unless, of course, we already have some experience with the chaos game.

The chaos game generalizes the behavioral rules to use the vertices of a regular n-gon as target points, and for each painter agent to move a fraction r of the distance to the randomly selected vertex. The specific case we've been working with has $n = 3$ and $r = 0.5$, but there are many other interesting possibilities. Many (but not all) of the possible combinations result in fractal sets. Unlike most other construction methods, the chaos game finds points in the fractal set in a random order.

Task 4: Modifying the model

Refining movement and display

1. There are a number of ways to change a turtle's location. So far, we've used the **`forward`** command primitive, which moves the turtle forward in the direction it's already facing. Another approach is to use the **`jump`** command primitive, which works in exactly the same manner as **`forward`**, except that NetLogo doesn't try to animate the travel of the turtle along its path.

   Modify your **`go`** procedure, to use **`jump`** instead of **`forward`**. What's the result?

2. If the model uses **`jump`**, it might not be that important for the turtles to appear at all. In NetLogo, each turtle has a **`hidden?`** variable, which we can set to **`true`** or **`false`**, to control whether a turtle is visible (alternatively, we can use the **`hide-turtle`** and **`show-turtle`** command primitives).

   Modify your **`setup`** procedure, to make all turtles invisible when they're created.

Task 5: Implementing the chaos game

Implementing the general chaos game

1. As noted above, the chaos game generalizes the behavioral rules to use the vertices of a regular n-gon as target points, and for each painter agent to move a fraction r of the distance to the randomly selected vertex. Consider how we might modify the model to support any n value in the set $\{3, 4, \ldots, 12\}$, and any r value in the set $\{0.01, 0.02, \ldots, 0.99\}$? (Hint: Where in our code do we see the number 3, representing the number of vertices? Can we replace that literal value with a variable that can be controlled in the user interface? Similarly, where do we see the number 0.5, representing the fraction of the distance to the target vertex that the painter turtle moves? Can we replace that literal value with a variable controlled in the user interface?)

2. Add two sliders (to control n and r) and modify the code as necessary to change the model into one that implements the general chaos game. Make sure you still get the expected results with $n = 3$, $r = 0.5$.

3. Experiment with different values of n and r. Can you find other interesting combinations of values?

References

[1] NetLogo Dictionary, Apr. 2015. [Online]. Available: http://ccl.northwestern.edu/netlogo/docs/index2.html. [Accessed: 24 Aug. 2015].

[2] Weisstein, Eric W. "Chaos Game", *MathWorld*, Mar. 2003. [Online]. Available: http://mathworld.wolfram.com/ChaosGame.html. [Accessed: 25 Aug. 2015].