In most computer programs (applications or services of one sort or another) there is only one thing going on at a time, and even if there are several things, the relationship between the computer and the user is simple, the interaction is through a single conceptual channel (the user interface).
When the user can write algorithms that can be executed on a computer then the user also becomes an author, and people can both observe the way the software runs, and make changes to it, active as designer and user at once. This relationship extends all the way from seasoned C++ engineers and their massive development environments, writing applications and systems software, to the humble macro programmer automating formatting or drafting operations in Word or AutoCad.
NetLogo is a development environment for designing and writing Logo programs, and provides a language and authoring environment (described in the user interface document).
However NetLogo is an example of parallel computing which means that Logo programs can be run in parallel (simultaneously) on the computer. This is actually impossible, since the Mac & PC both have just one processor, so NetLogo makes use of some nifty programming to do the thousands of calculations so fast that to the user they seem to be executing simultaneously.
Now, in order for the trick to work, the programs must not only be executed quickly, but they have to be orchestrated so that they all complete execution at the same time. This requires:

- an observer who can "see" all the programs and data at once
- a global clock which ticks once each time all the programs have run
- a limbo world which exists as a frozen snapshot of the world at a particular tick of the clock
- things which do the actual computation

**COMPONENTS of NETLOGO**

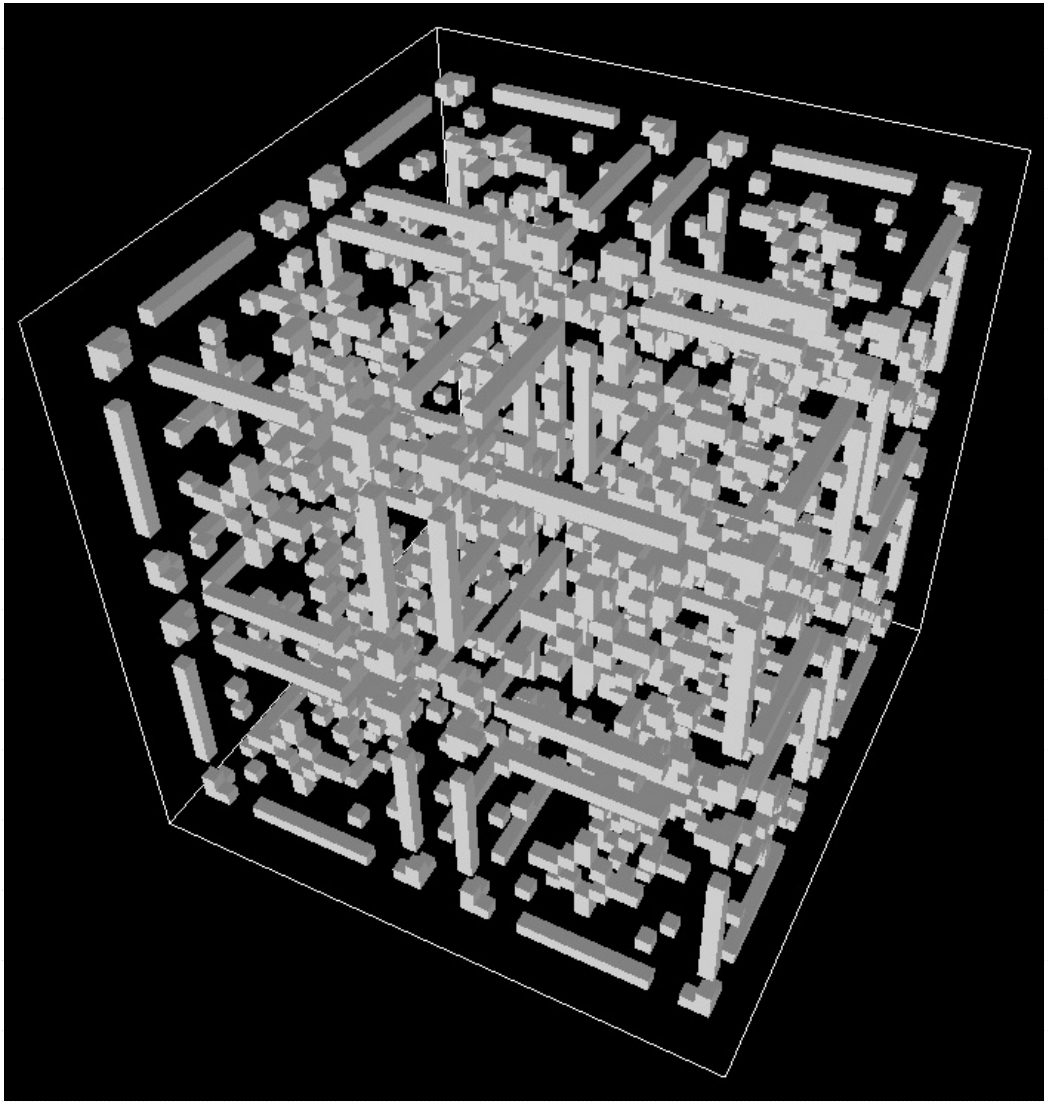
(the reference manual covers this section)
NetLogo itself is a relatively new development from a purely Mac based program called StarLogo, which divided the three components' codes (see below) into three separate scripting panes. NetLogo has has simplified the interface into a Procedures window and an Interface, where the only remains of the once complicated StarLogo is the Command Center scripting box from where the Observer can interactively write commands to the other two components.
Additionally, NetLogo allows to share scripts interactively via a hub and let's you create Java applets ready for upload to the web.

**The observer**

In NetLogo the observer has a defined status, rather than being the only possible actor as with conventional algorithms. The observer reports on the global emergent form of the parallel interactions of the patches and turtles. The observer can't talk directley to individual patches or turtles, and can only set the scene and report on the outcome.

**Patches**

NetLogo provides a cubic array of cells which we will be using for our "magic sponge". Each cell can have a state, a colour, and can interrogate its immediate neighbours to count up the amount of these pieces of information . Patches can alter their own internal states, and can diffuse state variables to



neighbours. They can find out about turtles standing on them, and turtles ca alter the patch state variables if they are standing on them.
NetLogo patches then, provide a programmable multi-state 3D CA, which has the added dynamics of agents (turtles) moving under their own rules through them.

**Turtles**

The original Logo (by Seymour Papert, MIT) consisted entirely of turtles (turtles allthe way down as the ancient cosmology has it) who could be programmed to move and draw lines by forward, left turn, right turn commands. NetLogo provides thousands of turtles who can not only do this, but can also interact with all other turtles, and patches.

Turtles on their own can be programmed to display emergent form in the same way as patches, the sphere example shows how global (observer) constraints and local ones can lead to stable states from random beginnings. The Reaction Diffusion example shows how patches interact locally to create global tiling (voronoi) structures.
One of the classic AL examples of emegent behaviour is the swarm algorithm which also uses just turtles.

**The User Interface window**

This contains the buttons needed to make the procedures you have written begin execution. For this project we need two buttons called "setup" and "go". The names of the buttons are the same as the names of the procedures. The go button is a "forever" button. The third component is the graphics window.
**todays project:- a Cellular Automaton**

the cells all have a state - a personal value - which can be 0 or 1. the state represents whether the cell is turned on or off. in the model the on cells are drawn, the off cells are left empty.

**the core of the algorithm** :: *the rules of transition* :

First we count up the states in our neighbourhood this value is stored in another variable called "howmany". it is on the basis of howmany that the cell decides whether to live or die.

```
;sum  the states of all neighbours (27 of them)

set howmany sum values-from neighbors3d [state]

;check whether of go live or die

      ifelse state = 0
            [if  howmany > 0 and howmany < 6 [set state 1] ]
            [if howmany >= threshold         [set state 0] ]
```

Governed by the ubiquitous 'conditional statement' *ifelse*, the rules of state transition determine the future state of the patch.

The *if* expresses the condition and the first set of brackets evaluate if the condition is ***true***. If the condition is ***false***, then the second set of brackets will be evaluated; in that case *else* is ***true***.

what's with the funky colours?
blue for built in language words
purple for supplied library routines
brown for constants (numbers)
black for your made up words

A cellular automaton is a model of a discrete dynamical system. Space, time and the states of the system are discrete.
Each point in a regular spatial lattice, called a cell, can have one of a finite number of states. The states of the cells in the lattice are updated according to a local rule. That is, the state of a given cell at a given time depends only on its own state one time step previously, and the states of its nearby neighbours at the previous time step.

**Self-help**

You will have noticed how some of the lines are indented whereas others aren't. The indent helps you to identify the structure of the program (i.e. lines that are dependent on a condition - also called nested - are wrapped at the top by an '[' and at the bottom by the ']'. Additionally, when you write some of your first lines attempt to comment those lines which are not self-evident. In NetLogo a line can be commented with the ';' as demonstrated in the algorithm above. Commented lines will not be read or executed by the program.
These techniques support you and others when trying to understand the program at a later date, or when the program in the future will become several thousand lines long.

Book: Michel Resnic,'Turtles, Termites and Traffic Jams'