Plotting

Eliot McIntire

Friday, September 12, 2014

Contents

1 Plotting in SpaDES

One of the major features of the SpaDES package is that can take advantage of the numerous visualization tools available natively or through user built packages (e.g., RgoogleVis, ggplo2, rgl). The main set of plotting functions that we have packaged with spades are built on top of the grid package. These allow for relatively fast plotting of rasters and points with the ability to make multi-frame plots without the module (or user) knowing which plots are already plotted. In other words, the main plotting function can handle modules that add plots, without them knowing what the current state of the active plotting device is. This means that the plotting can be also treated as modular. Furthermore, conventional R plotting still works, so you can use the features provided in this package or you can use base plotting functions without having to relearn a completely new set of plotting commands. This is called with the workhorse function, Plot, i.e., capital P.

First, we load some maps and put them into a stack. These maps are randomly generated maps that come with the SpaDES package.

```
#  Make list of maps from package database to load, and what functions to use to load them
library(SpaDES)
fileList <-
    data.frame(files =
     dir(file.path(
                   find.package("SpaDES",
                                lib.loc=getOption("devtools.path"),
                                quiet=FALSE),
```

```
                    "maps"),
         full.names=TRUE, pattern= "tif"),
     functions="rasterToMemory",
     .stackName="landscape",
     packages="SpaDES",
     stringsAsFactors=FALSE)

# Load files to memory (using rasterToMemory) and stack them (because .stackName is provided above)
loadFiles(fileList=fileList)
```

```
## Warning: Global parameters .stackName are not used in any module.
```

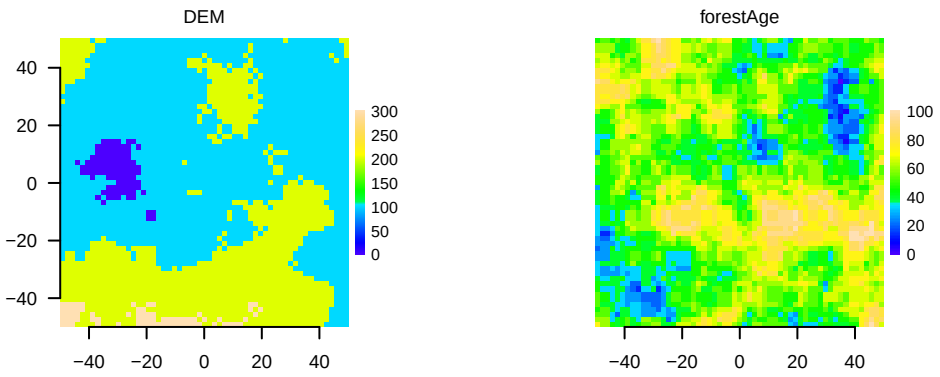
```
# extract a single one of these rasters
DEM <- landscape$DEM
#'
```

2 The Plot function

2.1 Layer types

There are several features of Plot that are worth highlighting. First, it can plot a mixture of RasterLayers, RasterStacks and SpatialPoints* objects. In the code snippet below, we create the list of files to load, which is every file in the "maps" subdirectory of the package. Then we load that list of files. Because we specified .stackName in the fileList, the loadFiles function will automatically put the individual layers into a RasterStack; the individual layers will not be available as objects within the R environment. If .stackNames did not exist, then the individual files would be individual objects.

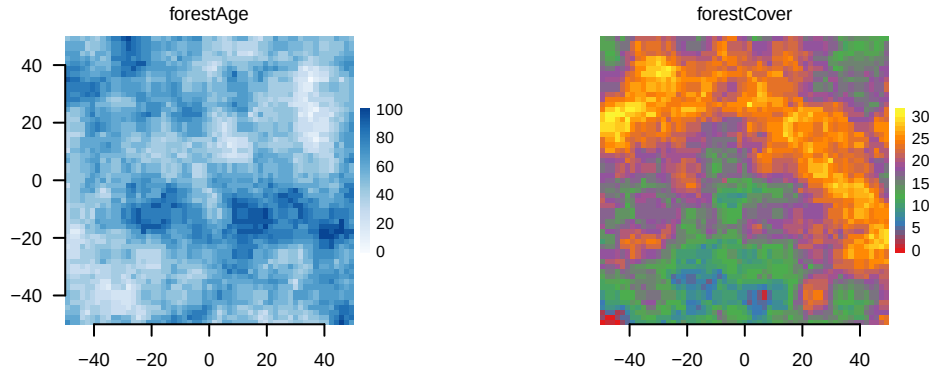
```
Plot(landscape[[1:2]])
```



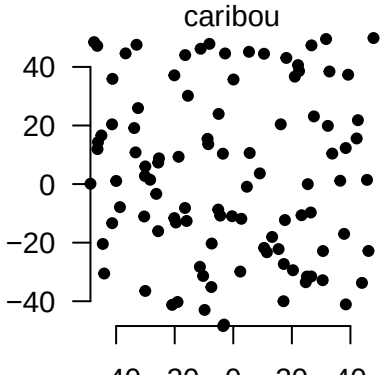
2.2 Colors

We likely won't want the default colors for every map. Currently, the best way to change the color of a map is to give it a colortable using the `setColors` function in `SpaDES`. Every RasterLayer can have a colortable, which gives the mapping of raster values to colors. If not already set in the file (many .tif files and other formats already have their colortable set), we can use setColors(Raster*) with a named list of hex colours, if a RasterStack, or just a vector of hex colors if only a single RasterLayer. These can be easily built with the RColorBrewer package, with the function brewer.pal(). But there are many other ways in R, see `colorRampPalette`.

```
# can change color palette
library(RColorBrewer)
setColors(landscape, n = 50)<-
              list(DEM=topo.colors(50),
                   forestCover=brewer.pal(9,"Set1"),
                   forestAge=brewer.pal("Blues",n=8),
                   habitatQuality=brewer.pal(9,"Spectral"),
                   percentPine=brewer.pal("GnBu",n=8))
Plot(landscape[[2:3]])
```



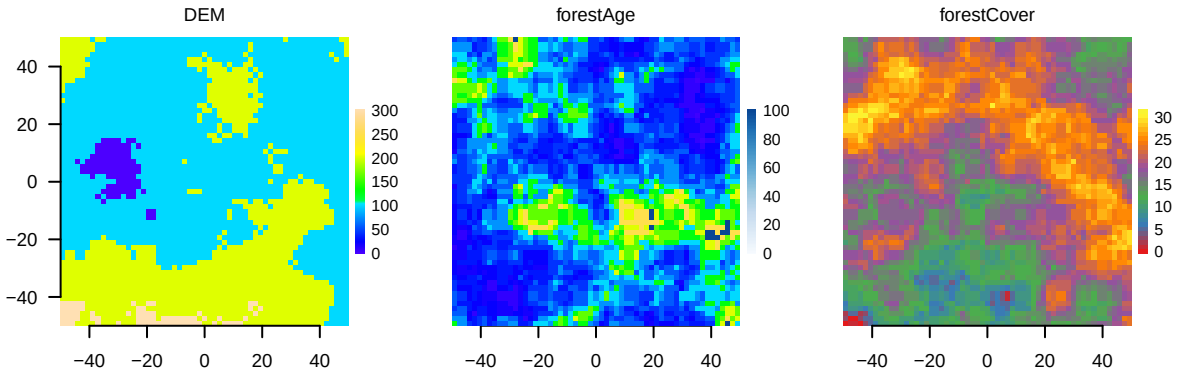

```
# make a SpatialPointsNamed object
caribou <- SpatialPointsNamed(coords=cbind(x=runif(1e2,-50,50),y=runif(1e2,-50,50)),
                              name="caribou")
Plot(caribou)
```



2.3 Names

It is critical in SpaDES plotting that every layer has a unique name. RasterLayers already have this functionality, contained within the element, `names`. RasterStacks do not, nor do SpatialPoints* objects. Names can be added to RasterLayers using `names` and to RasterStacks or SpatialPoints using the assignment functions `name` (in the form `names(Layer)<-"something"` or `name(Layer)<-"something"`). This would be necessary when a new Raster is created (say in a simulation) or if a new Raster is derived from another Raster, as the new one would inherit the original name. The new layer would then overplot the original layer, which is not the desired behavior.

```
#Make a new raster derived from a previous one; must give it a unique name
habitatQuality2 <- ((landscape$forestAge) / 100 + 1) ^ 6
#setColors(habitatQuality2) <- heat.colors(50)
Plot(landscape[[1:3]])
Plot(habitatQuality2, add=TRUE)
```

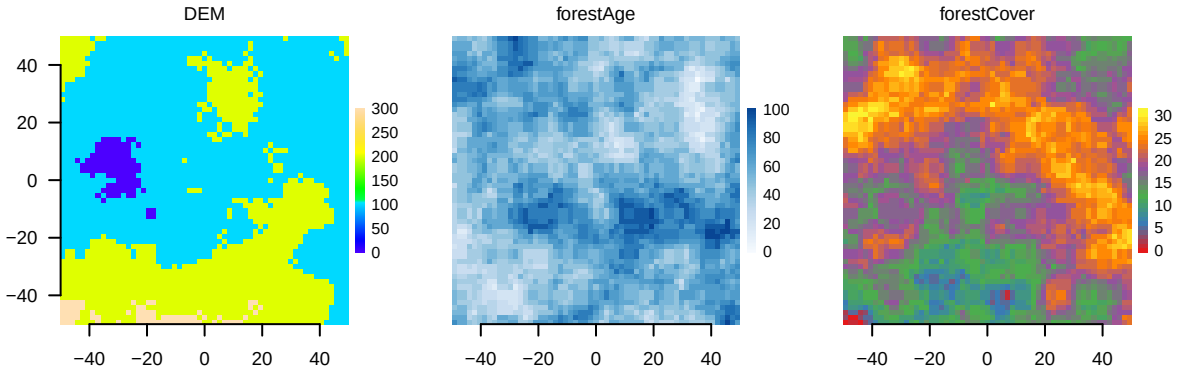


To get the correct behavior, give the new layer a unique name:

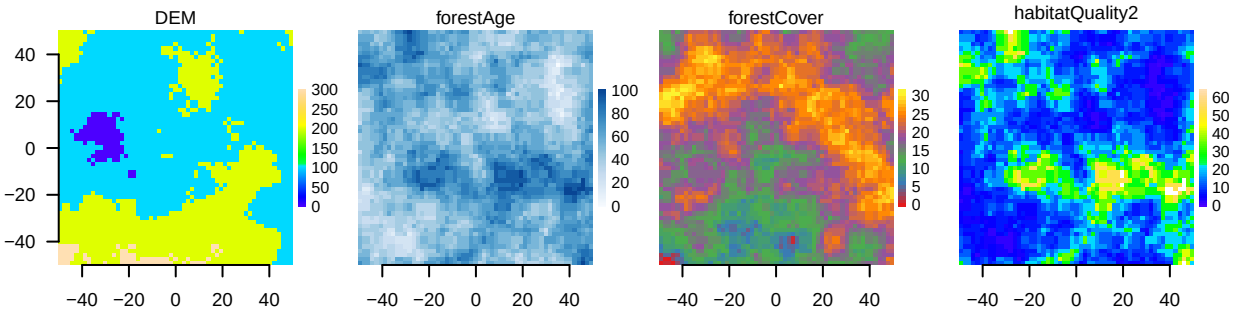
```
names(habitatQuality2) <- "habitatQuality2"
print(habitatQuality2)
```

```
## class       : RasterLayer
## dimensions  : 100, 100, 10000  (nrow, ncol, ncell)
## resolution  : 1, 1  (x, y)
## extent      : -50, 50, -50, 50  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : in memory
## names       : habitatQuality2
## values      : 1, 64  (min, max)
```

```
Plot(landscape[[1:3]])
```



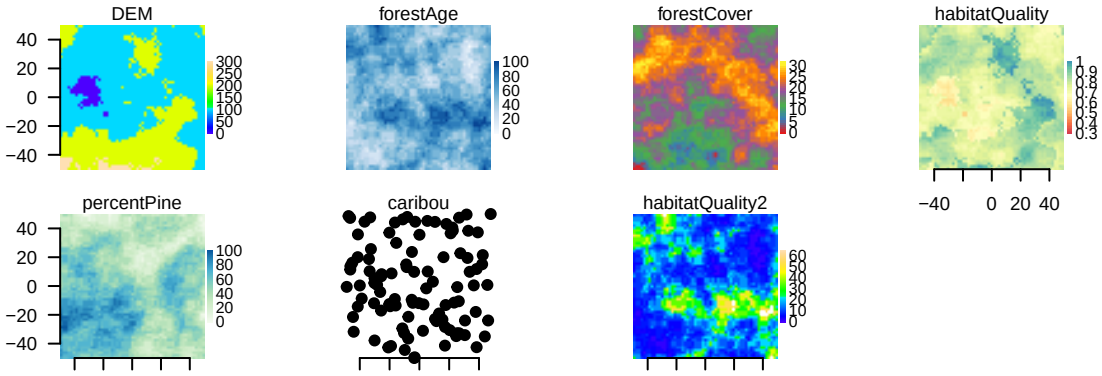

```
Plot(habitatQuality2, add=T)
```



2.4 Mixing Layer Types

Any combination of RasterStacks, RasterLayers, and SpatialPoints* objects can be plotted.

```
Plot(landscape, caribou, habitatQuality2)
```



3 Modularity

One of the main purposes of the `Plot` function is modularity. The goal is to enable any `SpaDES` module to be able to add a plot to the plotting device, without being aware of what is already in the plotting device. To do this, there is a hidden global variable (a `.arr` object of S4 class, "arrangement") created when a first Plot function is called. This object keeps the layer names, their extents, and whether they were in a RasterStack (and a few other things). So, when a new `Plot` is called, and `add` is used, then it will simply add the new layer. There may not be space on the plot device for this, in which case, everything will be replotted in a new arrangement, but taking the original R objects. This is different than the grid package engine for replotting. That engine was not designed for large numbers of plots to be added to a region; it slows down immensely as the number of plots increases.

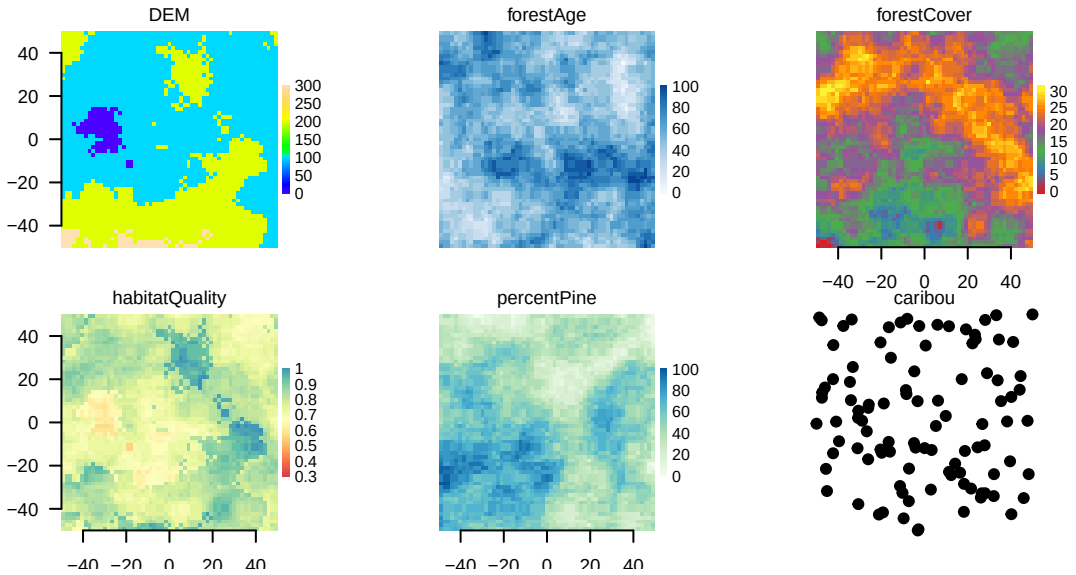
3.1 The `add` argument

There are essentially 3 types of adding that are addressed by this argument. Adding a new plot with:

3.1.1 a new name to a device with enough space

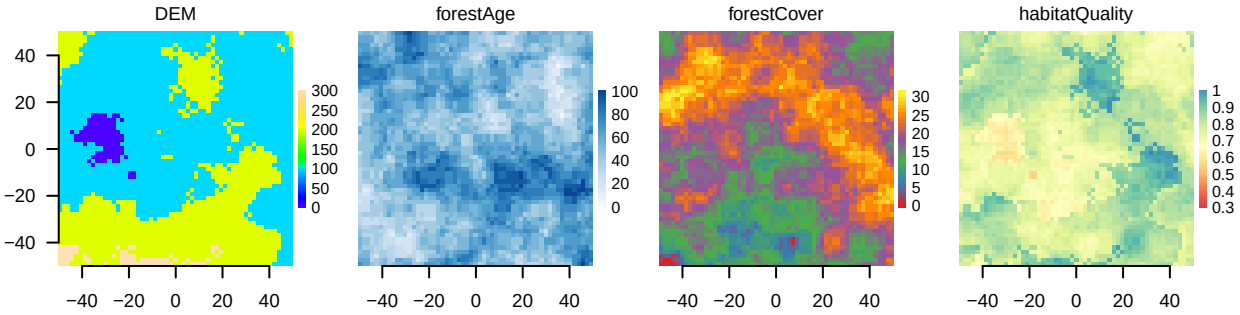
The `Plot` function simply adds the new plot in the available empty space. ### a new name to a device without enough space The `Plot` function creates a new arrangement, keeping the pre-exising order of plots, and adding the new plots afterwards. The plots will all be a little bit smaller, and in a different location on the device. ### a pre-existing name to a device The `Plot` function will overplot the new layer in the location as the layer with the same name. If colors in the layer are not transparent, then this will effectively block the previous plot. *This will automatically set legend, title and axes to FALSE.*

```
Plot(landscape)
# can add a new plot to the plotting window
Plot(caribou, add=T, axes=FALSE)
```

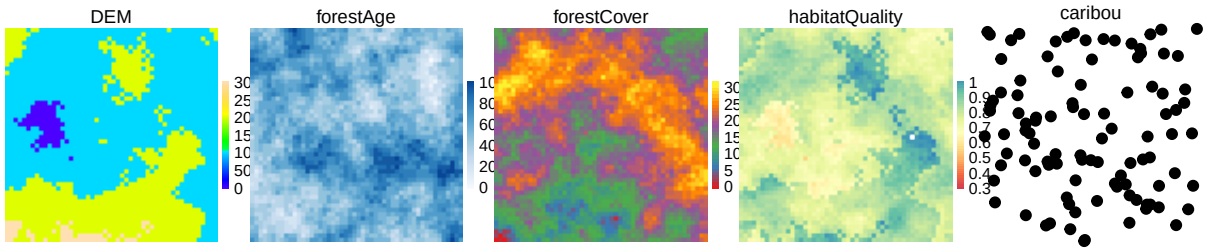


When there is not enough room on the plot, they will be rearranged and rescaled appropriately:

```
Plot(landscape[[1:4]])
```




```
# can add a new plot to the plotting window
Plot(caribou, add=T, axes=FALSE)
```



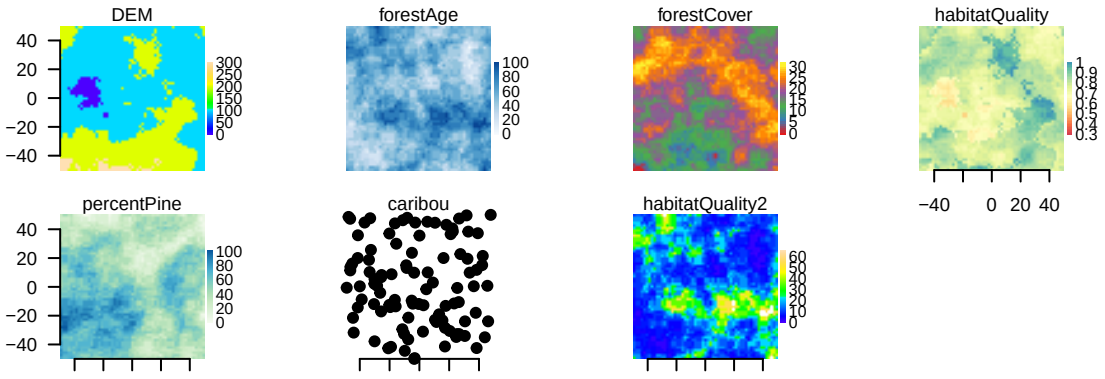
4 Plotting Speed

A second main purpose of the `Plot` function is to plot as fast as possible so that visual updates, which may be frequent, take as little time as possible. To do this, several automatic calculations are made upon a call to `Plot`. First, the number of plots is compared to the physical size of the device window. If the layers are RasterLayers, then they are subsampled before plotting, automatically scaled to the number of pixels that would be discernible by the human eye. See below for using the **`speedup`** argument.

4.1 `speedup`

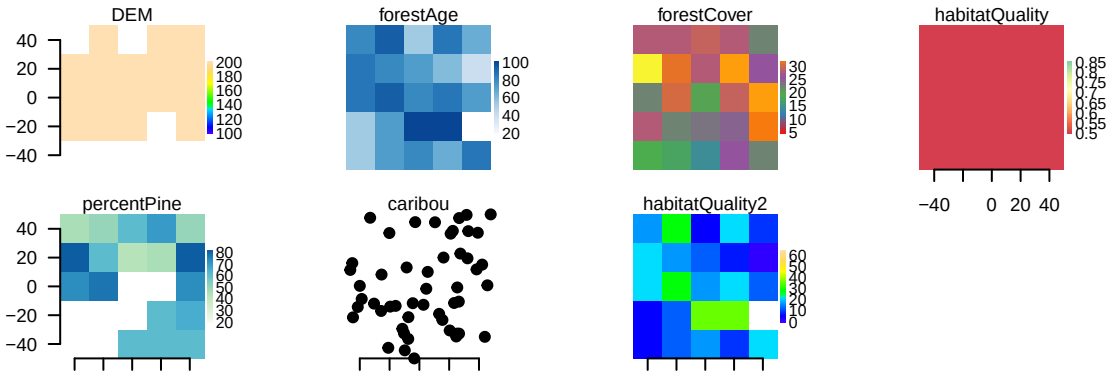
The argument, **`speedup`**, is a relative speed increase at the cost of resolution if it is >1. If it is between 0 and 1, it will be a relative speed decrease at the gain of resolution. This may be used successfully when the layer is particularly coarse resolution. The speedup gains here are modest because the rasters are relatively small (10,000 pixels), but will be much greater for larger rasters. For SpatialPoints, the default is to only plot 10,000 points; if there are more than this in the object, then a random sample will be drawn. Speedup is used as the denominator to determine how many to plot `10000/speedup`. :

```
system.time(Plot(landscape, caribou, habitatQuality2))
```




```
##    user  system elapsed
##    0.35    0.00    0.34
```

```
system.time(Plot(landscape, caribou, habitatQuality2,speedup=200))
```



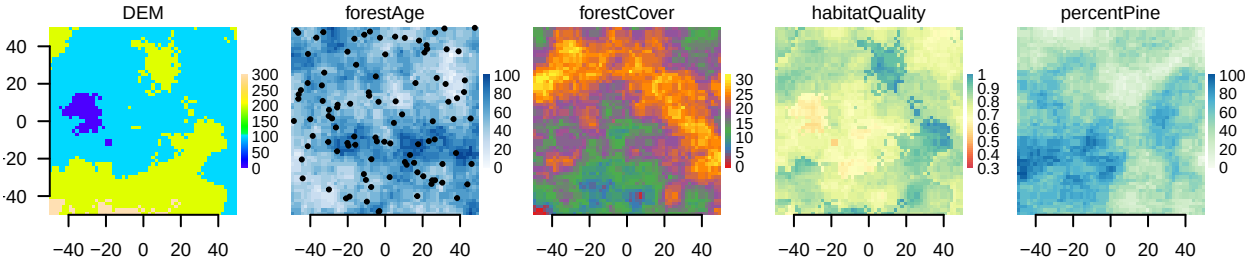

```
##    user  system elapsed
##    0.28    0.00    0.28
```

```
# can add a new plot to the plotting window
```

5 Overplotting: addTo

There are times when it is useful to add a plot to a different plot with a different name. In these cases, the `add` argument will not work. The argument `addTo` will allow plotting of a RasterLayer or SpatialPoints* object on top of a RasterLayer, *that does not share the same name*. This can be useful to see where agents are on a RasterLayer, or if there is transparency on a second RasterLayer, it could be plotted on top of a first RasterLayer.

```
Plot(landscape)
Plot(caribou, addTo="forestAge", size=2, axes=F)
```




```
#'
```

There are several situations that do not plot. A call to Plot where there are two RasterLayers with the same name will return an error. This is true even if one of the layers is in a RasterStack, so is not explicitly named in the call to `Plot`.

```
Plot(landscape, caribou, DEM)
```