Plotting with SpaDES

Eliot J. B. McIntire

January 08 2015

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, ggplot2, rgl). Nevertheless, none of these was built to be fast, modular, and replottable hundreds, thousands or more times during a simulation model. We therefore built a plotting function to fulfill this need. The main plotting function, Plot (*i.e.*, with a capital P), is built using the grid package. We have specifically built a plotting system that allows for relatively fast plotting of rasters, points, and polygons 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 SpaDES modules, each of which can add plots, without each knowing what the current state of the active plotting device is. This means that plotting can be treated as modular. Importantly, conventional R plotting (*e.g.*, plot, hist, etc.) still works fine, so you can use the features provided in this package or you can use base plotting functions without having to relearn a new set of plotting commands. The Plot function is therefore intended to be used as a way to interact visually during model development. If fine

tuning and customization are desired, other plotting tools may be more suited (*e.g.*, `ggplot2`, or a dedicated GIS program).

To demonstrate plotting, we first load some maps. These maps are randomly generated maps that come with the `SpaDES` package. 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, therefore, not be available as individual objects within the R environment. If `.stackNames` did not exist, then the individual files would be individual objects.

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

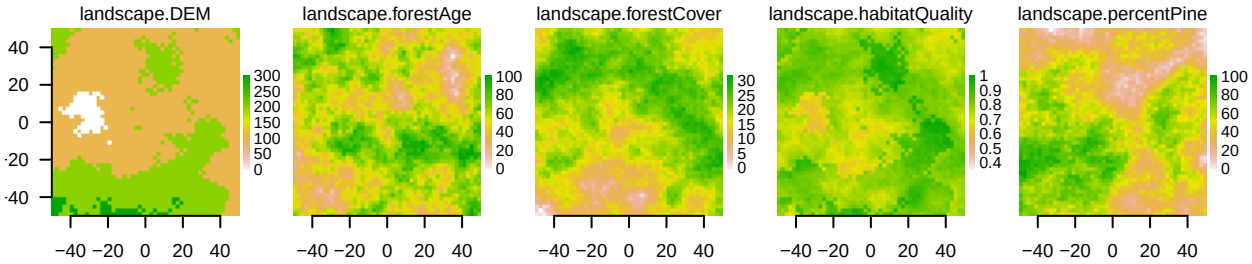
2 The `Plot` function

There are several features of `Plot` that are worth highlighting.

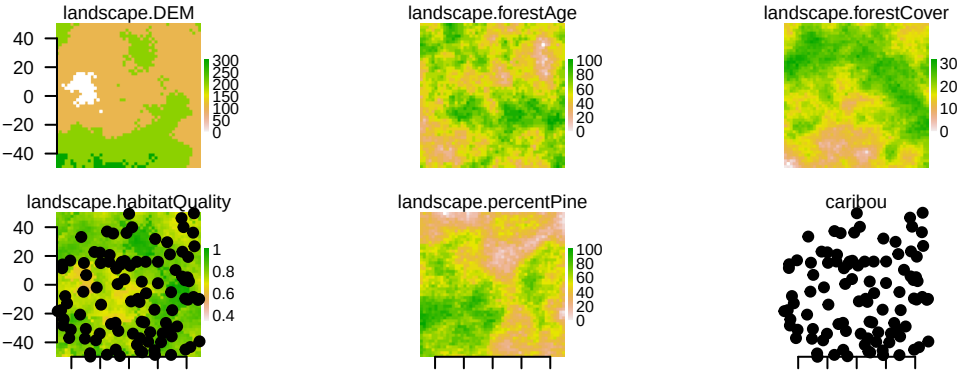
2.1 Layer types

Its main purpose is to plot spatial type objects. Specifically, it currently can plot RasterLayers, RasterStacks, SpatialPoints, *and SpatialPolygons* objects. Because `Plot` uses the `grid` package, changing plot parameters is with the `gp=gpar()` designation. See `?gpar` for options.

```
Plot(landscape, new=TRUE)
```

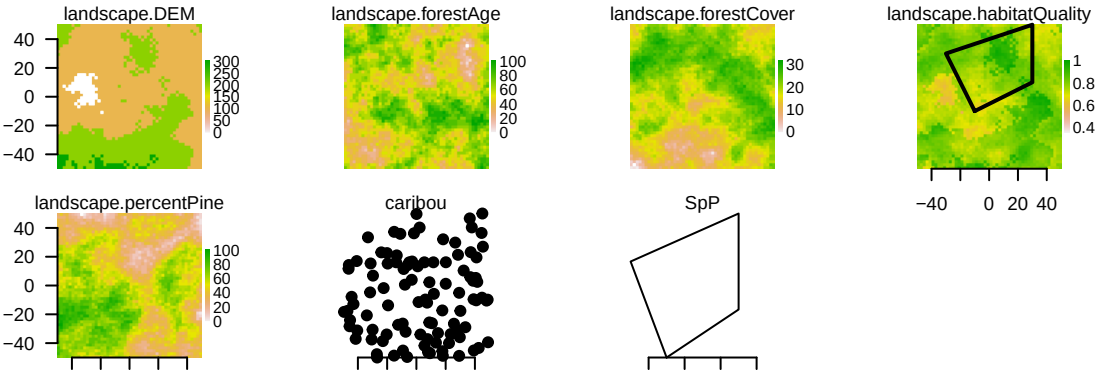


```
# make a SpatialPoints object
caribou <- SpatialPoints(coords=cbind(x=runif(1e2,-50,50),y=runif(1e2,-50,50)))
Plot(caribou)
Plot(caribou, addTo="landscape.habitatQuality")
```




```
# SpatialPolygons
Sr1 = Polygon(cbind(c(2,4,4,1,2),c(2,3,5,4,2))*20-50)
Sr2 = Polygon(cbind(c(5,4,2,5),c(2,3,2,2))*20-50)

Srs1 = Polygons(list(Sr1), "s1")
Srs2 = Polygons(list(Sr2), "s2")
SpP = SpatialPolygons(list(Srs1,Srs2), 1:2)
Plot(SpP)
Plot(SpP, addTo="landscape.habitatQuality", gp=gpar(lwd=2))
```

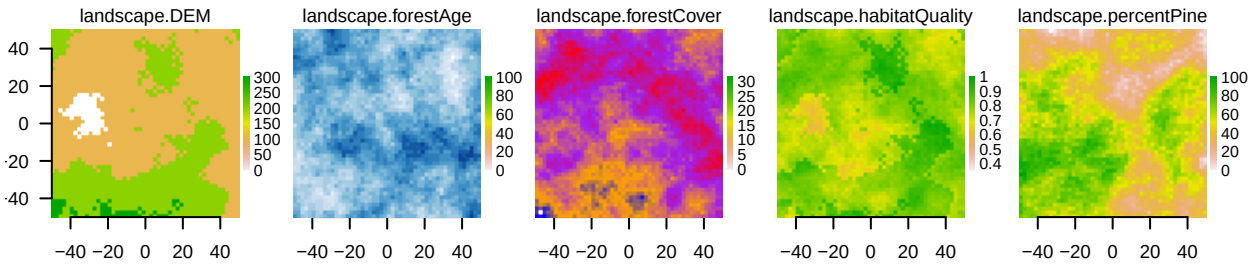


2.2 Colors

We likely won't want the default colors for every map. There are two ways to change the color of a map: by adding a `colortable` to a `Raster*` object (*e.g.*, using the `setColors` function in `SpaDES`), or using the `cols` argument. Adding a colortable to a `Raster*` will be more persistent (*i.e.*, it will stay with the same color table between calls to `Plot`) than using an argument in the `Plot` function. 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()`, or `colorRampPalette` or `heat.colors()` etc. Note that overplotting will not overplot the legend; in general, overplotting should be used for cases where the maps are compatible with the underlying map layer. See overplotting below.

`zero.color` is an optional string indicating the color for zero values, when zero is the minimum value, otherwise, it is treated as any other color. Default transparent. Use NULL if zero should be the value given to it by the colortable associated with the Raster. See text about `zero.color` below after raster values have changed.

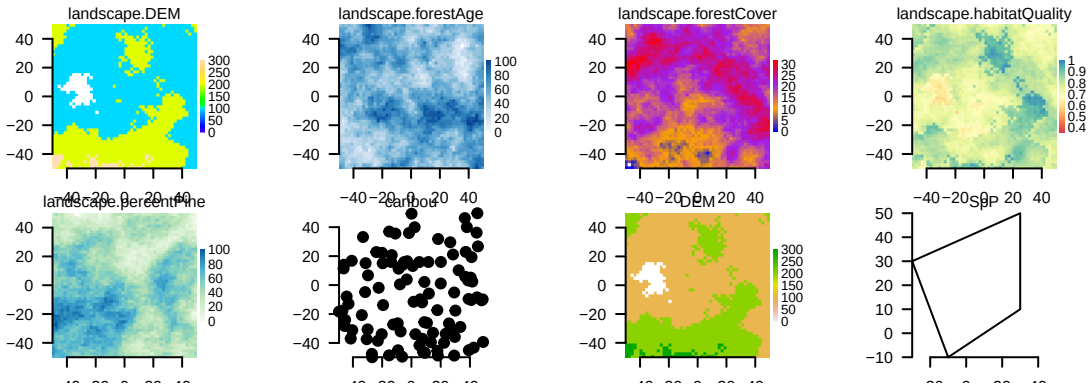
```
# can change color palette
Plot(landscape, new=TRUE)
library(RColorBrewer)
setColors(landscape, n=50) <-
              list(DEM=topo.colors(50),
                   forestCover=colorRampPalette(c("blue","orange","purple","red"))(50),
                   forestAge=brewer.pal("Blues", n=8),
        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



2.3 Mixing Layer Types

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

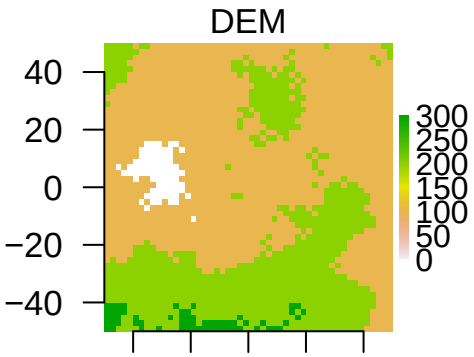
```
Plot(landscape, caribou, DEM, SpP, new=TRUE, axes=TRUE, gp=gpar(cex=0.5))
```



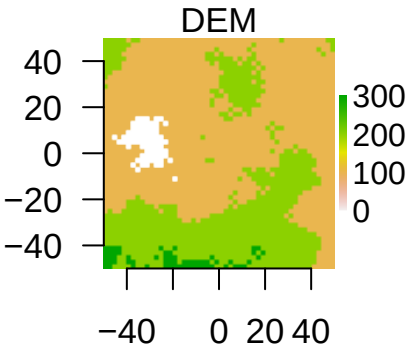
2.4 `visualSqueeze`

Under most circumstances, the plotting regions will be automatically scaled to maximize the area taken by the map layers, minimizing white space, but allowing axes, legends and titles to be visible when they are plotted. In some devices, this automatic scaling is imperfect, so axes or legends may be squished. The `visualSqueeze` argument is an easy way to shrink or grow the plots on the device. The default value is 0.75 representing ~75% of the area. If the plots need to be slightly smaller, this could be set to 0.6; if they can be larger, visualSqueeze could be set to 0.8.

```
# x axis gets cut off in pdf and html
Plot(DEM, new=TRUE)
```




```
Plot(DEM, visualSqueeze=0.6, new=TRUE)
```



A key reason why the legends or axes are cut off sometimes is because there is a minimum threshold for font size for readability. So, either visualSqueeze can be set or making a larger device will usually also solve these problems.

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 `.spadesArrN` [where N is the device number] 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 `new` 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.

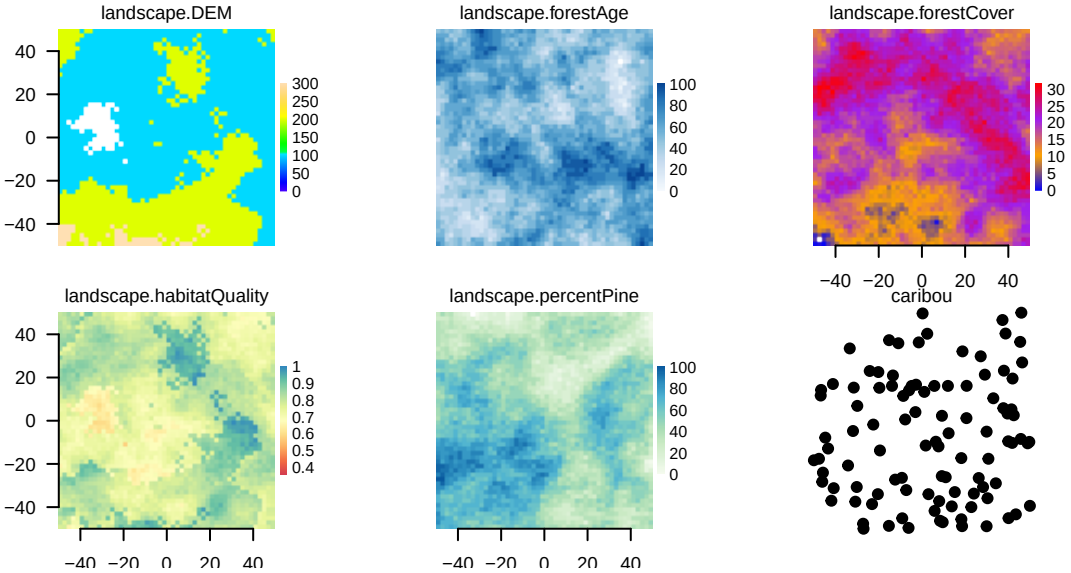
3.1 The `new` argument

There are essentially 3 types of adding that are addressed by this argument, 1) adding a new plot with enough empty space to accommodate the new plot, 2) without this empty space, and 3) where the device already has a pre-existing plot of the same name.

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.

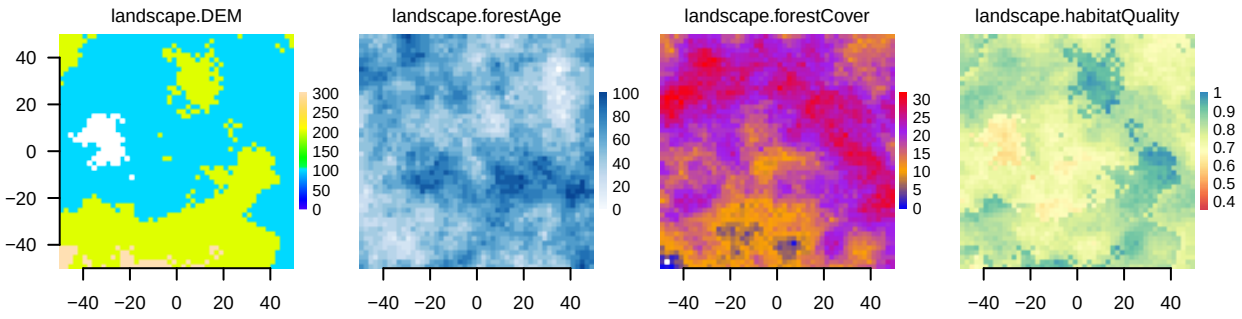
```
Plot(landscape, new=TRUE)
# can add a new plot to the plotting window
Plot(caribou, new=FALSE, axes=FALSE)
```



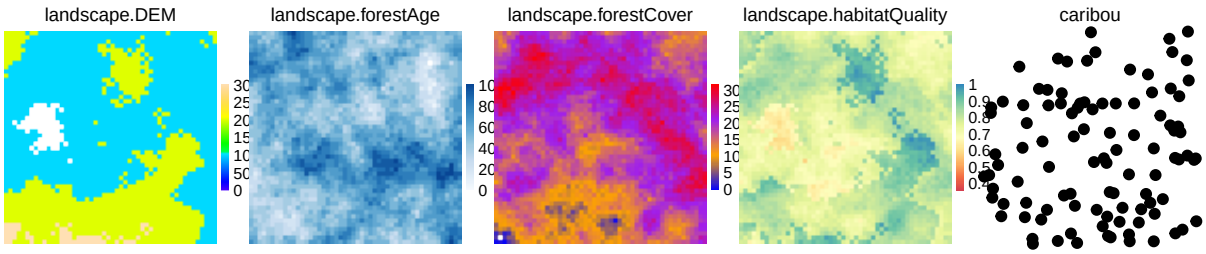
3.1.2 a new name to a device without enough space

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 (assuming the device has not changed size), and they will be in different locations on the device.

```
Plot(landscape[[1:4]], new=TRUE)
```



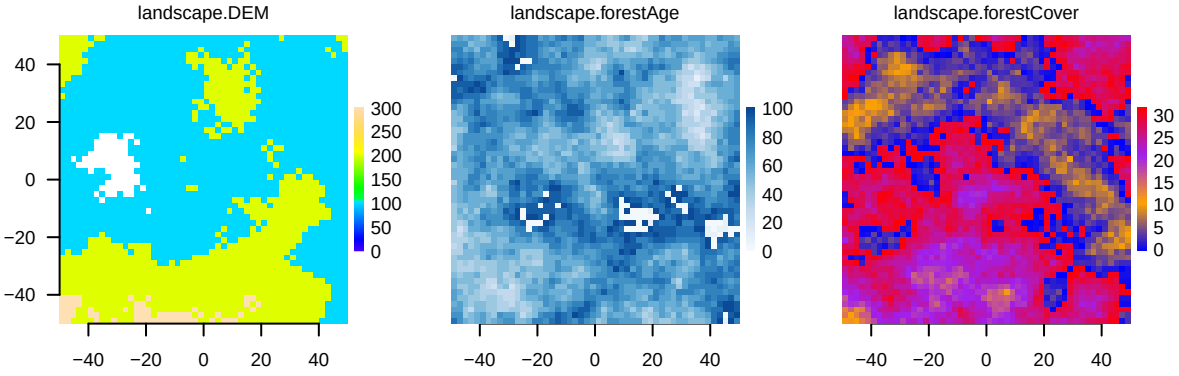

```
# can add a new plot to the plotting window
Plot(caribou, new=FALSE, axes=FALSE)
```



3.1.3 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[[1:3]], new=TRUE)
landscape$forestAge[] = ((landscape$forestAge[] + 10) %% 100)
landscape$forestCover[] = ((landscape$forestCover[] +10) %% 30)
# can add a new plot to the plotting window
Plot(landscape[[2:3]], new=FALSE)
# note that zeros are treated as no color by default. If this is not the correct behavior, use
#  zero.color=NULL
Plot(landscape[[2:3]], new=FALSE, zero.color=NULL)
```



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. If the layer is a SpatialPoints* object, then a maximum of 10,000 points will be plotted. These defaults can be adjusted by using the `speedup` argument. Broadly, `speedup` is a number >0, where the default is 1. Numbers >1 will plot faster; numbers between 0 and 1 will plot slower. See below for using the `speedup` argument.

4.1 `speedup`

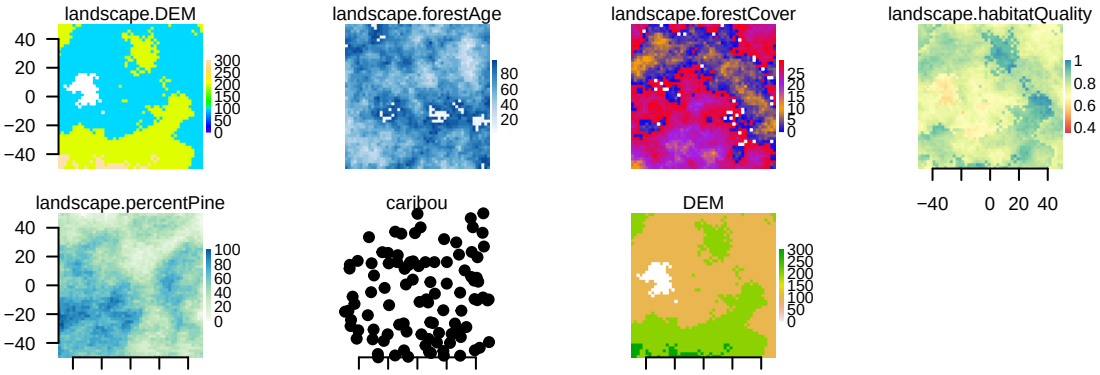
The `speedup` argument is a *relative* speed increase at the cost of resolution if it is >1. If it is between 0 and 1, it will be a relative speed decrease at the gain of resolution. This may be used successfully when the layer

texture is particularly coarse, *i.e.*, there are clusters of identical pixels, so subsampling will have little effect. In the examples below, the speedup gains are modest because the Rasters are relatively small (10,000 pixels). This speed gain 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`.

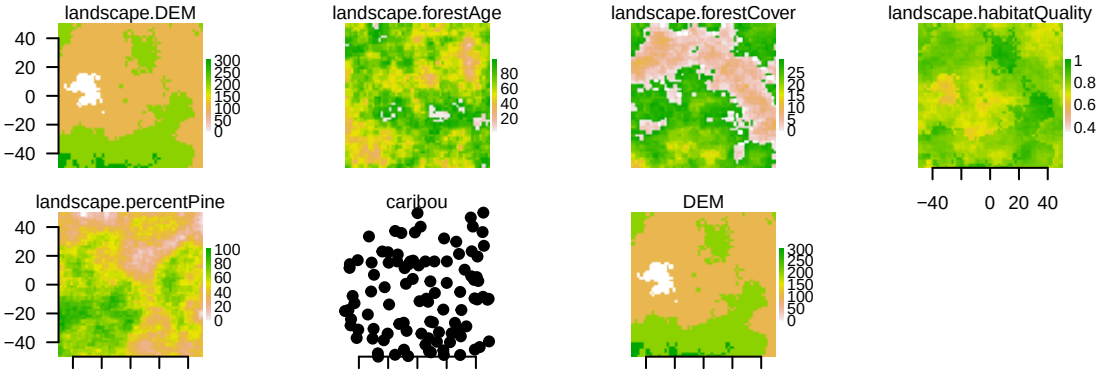
In the example here, the speedup actually slows down plotting because the rasters are already very small. This would not be the case when the original Raster had 1e8 pixels.

```
system.time(Plot(landscape, caribou, DEM, new=TRUE))
```




```
##    user  system elapsed
##   0.130   0.006   0.134
```

```
system.time(Plot(landscape, caribou, DEM, speedup=10, new=TRUE))
```




```
##    user  system elapsed
##   0.429   0.018   0.442
```

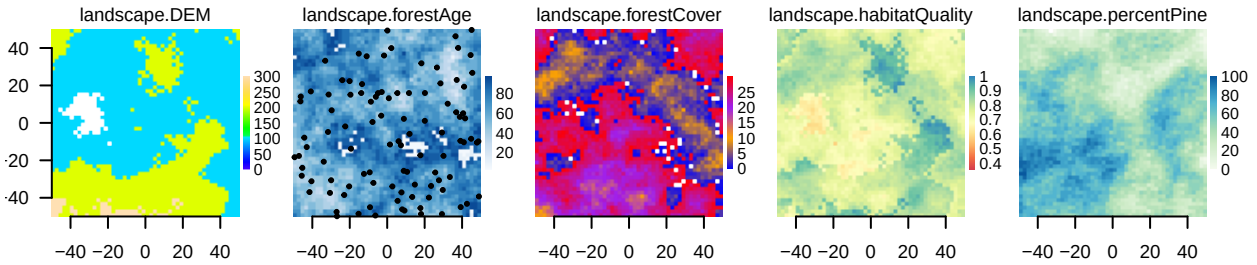
```
# can add a new plot to the plotting window
```

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



6 Using RStudio Plots window

The built in RStudio Plot window is particularly slow. It is recommended to always create a new plotting device whenever real simulations are being done and they will be substantially faster. This may change in a future version of RStudio. Until then, we have created a function, `dev` which will add devices up to the number in the parenthesis, or switch to that device if it is already open. If an RStudio plot has not been called, dev(2) will create a new device outside RStudio. If a plot has already occurred in RStudio's embedded plot window, then dev(4) will create a new device outside RStudio. dev(4) on its own will either create 3 new devices (device numbers 2, 3 and 4 because device number 1 is never used in R), or 1 new device.

```
# simple:
dev(4)

# better:
#Plot all maps on a new plot windows - Do not use RStudio window
if(is.null(dev.list())) {
   dev(2)
} else {
 if(any(names(dev.list())=="RStudioGD")) {
   dev(which(names(dev.list())=="RStudioGD")+3)
 } else {
   dev(max(dev.list()))
 }
}
```

7 Interacting with plots

7.1 `clickValues`

This can be used to obtain the values on the plotting device at the locations of the mouse clicks. This will work on multiplanel plots. Note that plotting of rasters with the grid package does not allow for partial pixels to be plotted at the edges of the raster. As a result, the edges of pixels may not perfectly line up with the coordinates that they appear with. *Do not rely on exact values when zoomed it*

```
Plot(landscape, new=TRUE)
clickValues(3) #click at three locations on the Plot device
```

7.2 clickExtent

This can be used like zoom for base package plot window. Click two corners of a Plot.

```
Plot(landscape, new=TRUE)
clickExtent() #click at two locations on the Plot device
```