Building modules in SpaDES

Alex M. Chubaty
Natural Resources Canada, Pacific Forestry Centre
email: achubaty@nrcan.gc.ca

Eliot McIntire
Natural Resources Canada, Pacific Forestry Centre
email: emcintir@nrcan.gc.ca

August 29, 2014

Contents

1 Introduction

1.1 Module overview

As described in the introductory vignette, `SpaDES` is event-driven, meaning that different actions are performed on data objects based on the order of scheduled events. A `SpaDEs` module describes the processes or activities that drive simulation state changes. Each activity consists of a collection of events which are scheduled depending on the rules of your simulation. Each event may evaluate or modify a simulation data object, or perform other operations such as saving and loading data objects or plotting.

A `SpaDES` module consists of a single `.R` source file, containing the code for 1) defining the event types described in the module, and 2) describing what happens during the processing of each event type.

1.2 Events

1.2.1 Simulation event list

The event queue is stored in a slot in a `simList` simulation object. Each event is represented by a `data.table` row consisting of the time the event is to occur (`eventTime`), the name of the module from which the event is taken (`moduleName`), and a character string for the programmer-defined event type (`eventType`). This list is kept sorted by `eventTime`, and events are processed in sequence beginning at the top of the list. Completed events are removed from the queue.

1.2.2 Module events

When a call to the event list is made, the event is processed by the module specified by `moduleName`. The module code then determines the event type and executes the code for that event. For each event type within a module: 1) the instructions for what happens for this event get executed; and 2) there is an optional call to `scheduleEvent`, which schedules a future event. A module can schedule other event types from within the same module, and should not call other modules because this introduces module dependencies, which breaks the "drop-in/replace" modularity of your simulation model.

1.2.3 Dependencies

Each module schedules its own events (e.g., a "fire" module may schedule "burn" events) and only uses its own data objects (or shared global objects). Modules that behave in this way are independent of one another, which is the preferred way to design and implement modules. Maintaining strict modularity allows the removal, additon, and replacement of modules without having to rewrite your code.

Module event dependencies complicate the construction of simulation models, and hinder the ability to develop and deploy models with modularity. If two modules are actually depedent on each others' events, then you should consider whether they really are separate modules or should be merged into a single module.

1.3 Objects

1.3.1 Data types

As you build your modules for your simulation, you can use any of R's data types to store your objects and data. In particular, matrices (including vectors) and lists work well for this purpose because, as of R version 3.1, they are more efficient, reducing your model's memory footprint and speeding up your code's execution. Other useful datatypes include `Raster*` and `SpatialPoints*` objects (see examples below).

1.3.2 Global objects

R passes copies of objects to functions (instead of using pass-by-reference), so the typical R function never acts directly on the global copy of the variable. Within a function, all operations are generally performed on a local copy of the object, which is then either discarded or returned back to the parent environment upon when the function completes. This object-copying behaviour has given R a reputation of being slow in comparison to languages like C and FORTRAN. However, recent improvements to R have made it more memory efficient and faster to execute, in part by minimizing the number of object copies made.

One way to reduce object copying is to work directly on objects in the global environment. Although this practice is not common among R users, it can be done. Use the superassignment operator (`<<-`) to assign global objects to reduce copying large objects (such as maps), which slows model execution. Altenatively (and more "correctly"), use the `assign` function to assign objects directly to the global environment. Likewise, use `get` to bring global objects into your functions.

1.3.3 Module object dependencies

As noted above, modules should not depend on one another for event scheduling. However, it is often useful to develop collections of modules that interact indirectly and are dependent on shared data objects. Modules can be designed to rely on outputs (data objects) from other modules. When objects are shared between modules, it is important to state these dependencies explicitly. To check whether global objects upon which a model depends exist, a call to `checkObject` is made during module initilization.

Note that modules need not be inter-dependent on one another: module B may depend on module A (for example to initialize a data object), without module A depending on module B.

1.4 Parameters

The parameter list in the `simList` object are used to pass parameters to modules. The nested named list structure allows passing as many parameters as needed for your simulation. We suggest passing a list of all the parameters needed for a single module together.

2 Default SpaDES modules

There are a number of built-in modules that provide useful functionality. These modules have their names prefaced with a dot to indicate they are "special".

2.1 .checkpoint module

Schedule automatic simulation checkpointing to allow you to resume a previously saved simulation. All objects in the global environment including the state of the random number generator are saved and can be reloaded. Checkpoint frequency and filename can be passed as parameters to the simulation object as illustrated below. By default, checkpointing is not used unless the `interval` parameter is provided; the simulation checkpoint file (if one exists) is loaded if the `file` parameter is specified.

```
> library("SpaDES")
> # initialize a new simulation, setting the checkpoint interval and checkpoint filename.
> times <- list(start=0, stop=100)
> outputPath=file.path("~", "tmp", "simOutputs")
> parameters <- list(.globals=list(.stackName="landscape", .outputPath=outputPath),
+                    .checkpoint=list(interval=10, file="chkpnt.RData"),
+                    randomLandscapes=list(nx=1e2, ny=1e2, inRAM=TRUE,
+                                          .plotInitialTime=0, .plotInterval=1e3))
> modules <- list("randomLandscapes")
> path <- system.file("sampleModules", package="SpaDES")
> mySim <- simInit(times=times, params=parameters, modules=modules, path=path)
> spades(mySim)

  |
  |                                                                      |   0%
  |
  |.......                                                               |  10%
  |
  |..............                                                        |  20%
  |
  |......................                                                |  30%
  |
  |.............................                                         |  40%
  |
  |.................................                                     |  50%
  |
  |.........................................                             |  60%
  |
  |.............................................                         |  70%
  |
  |.................................................                     |  80%
  |
  |.....................................................                 |  90%
  |
  |.....................................................................| 100%

> # retrieve the checkpoint params from the simulation object
> simParams(mySim)$.checkpoint

$interval
[1] 10
```

```
$file
[1] "chkpnt.RData"

> simParams(mySim)$.checkpoint$interval

[1] 10

> simParams(mySim)$.checkpoint$file

[1] "chkpnt.RData"
```

2.2 .progress module

Schedule updates to the simulation progress bar.

```
> # initialize a new simulation, setting the progress parameters
> mySim <- simInit(times=list(start=0.0, stop=100),
+                  params=list(.progress=list(.graphical=FALSE, .progressInterval=10),
+                              randomLandscapes=list(nx=1e2, ny=1e2, inRAM=TRUE)),
+                  modules=list("randomLandscapes"),
+                  path=system.file("sampleModules", package="SpaDES")
+ )
> # retrieve the checkpoint params from the simulation object
> simParams(mySim)$.progress
> simParams(mySim)$.progress$.graphical
> simParams(mySim)$.progress$.progressInterval
```

2.3 .load and .save modules

Schedule object save and file load events by passing parameters to the save and load parameters for each module. Unlike the default modules above, loading and saving is designed to be scheduled by the user from within a module.

Loading files can be done be passing a simulation parameter called `.loadFileList` which can be a `list` or a `data.frame` and consists minimally of a column called `files` listing the filepaths of the files to be loaded. Additional columns can also be provided:

- `objs`: a character string indicating the name of the object once the file is loaded.
- `funs`: a character string indicating the function to be used to load the file.
- `intervals`: a numeric indicating the interval between repeated loading of the same file. This should be `NA` or the column absent if the file is only loaded once.
- `loadTime`: a numeric indicating when the file should be loaded. Defaults to `simTime = 0`, but this can be any time. The loading will be scheduled to occur at the "loadTime", whatever that is. If the same file is to loaded many times, but not at a regular interval, then there should be separate line, with a unique loadTime for each.
- `args`: a list of lists of named arguments, one list for each loading function. For example, if raster is a loading function, `args = list(native = TRUE)`. If there is only one list, then it is assumed to apply to all load attempts and will be repeated for each load function.

Saving objects to file can be done by passing the appropriate arguments as parameters. The key values to include are:

- `.saveObjects`: a character vector naming the objects to be saved.
- `.savePath`: A path to which the object will be saved.
- `.saveInitialTime`: the time at which the first save is scheduled.
- `.saveInterval`: the interval at which objects will be saved. Used to schedule save events.

```
> # initialize a new simulation, setting the load and save parameters
> filelist <- file.path(find.package("SpaDES", quiet=FALSE),"maps")
> mySim <- simInit(times=list(start=0.0, stop=100),
+                  params=list(
+                    .loadFileList=data.frame(files=filelist, stringsAsFactors=FALSE),
+                    randomLandscapes=list(nx=1e2, ny=1e2, inRAM=TRUE,
+                                .saveObjects=c("habitat"),
+                                .savePath=file.path("output", "randomLandscapes"),
+                                .saveInitialTime=0, .saveInterval=10)
+                    ),
+                  modules=list("randomLandscapes"),
+                  path=system.file("sampleModules", package="SpaDES")
+ )
> # retrieve the load and save params from the simulation object
> simObjectsLoaded(mySim) # shows what's been loaded
> simFileList(mySim) # returns empty if objects successfully loaded
> simParams(mySim)$randomLandscapes$.saveObjects
> simParams(mySim)$randomLandscapes$.savePath
> simParams(mySim)$randomLandscapes$.saveInitialTime
> simParams(mySim)$randomLandscapes$.saveInterval
> # schedule a recurring save event [WITHIN A MODULE]
> nextSave <- simCurrentTime(sim) + simParams(sim)$randomLandscapes$.saveInterval
> sim <- scheduleEvent(sim, nextSave, "randomLandscapes", "save")
```

3 Creating a new module

3.1 Module design considerations

perhaps a bit about design phlosophy, relating back to dependencies and how to carefully build modules that actually retain modularity.

As noted above, modules should function as independently as possible so that they retain their modularity. While it may be useful for modules to exhibit indirect dependence on each other via shared data objects (such as maps), modules should not depend directly on each other via event scheduling.

3.2 Using the module template

Code for new modules can be developed quickly using the template generator function `newModule` which accepts as arguments the name of the new module, a directory path in which to create the new module, and a logical indicating whether to open the newly created module code file for editing.

```
> # create a new module called "randomLandscape" in the "custom-modules" subdirectory
> # and open the resulting file immediately for editing.
> newModule(name="randomLandscapes", path="custom-modules", open=TRUE)
```

The newly created file can now be modified in the identified sections and customized to your module. However, it is very important that you do not edit portions of the file outside of the designated sections or your module may fail to work properly within a simulation.

3.2.1 Plotting

Plotting events can also be scheduled similarly to save events, by passing module-specific parameters indicating when to schedule the first plot event and how often to rescheduling recurring plot events.

- `.saveObjects`: a character vector naming the objects to be saved.
- `.savePath`: A path to which the object will be saved.
- `.saveInitialTime`: the time at which the first save is scheduled.
- `.saveInterval`: the interval at which objects will be saved. Used to schedule save events.

```
> # initialize a new simulation, setting the load and save parameters
> mySim <- simInit(times=list(start=0.0, stop=100),
+                  params=list(
+                    randomLandscapes=list(nx=1e2, ny=1e2,
+                                .plotInitialTime=0, .plotInterval=1)
+                    ),
+                  modules=list("randomLandscapes"),
+                  path="SAMPLE"
+ )
> # retrieve the plotting params from the simulation object
> simParams(mySim)$randomLandscapes$.plotInitialTime
> simParams(mySim)$randomLandscapes$.plotInterval
> # schedule a recurring save event [WITHIN A MODULE]
> nextPlot <- simCurrentTime(sim) + simParams(sim)$randomLandscapes$.plotInterval
> sim <- scheduleEvent(sim, nextPlot, "randomLandscapes", "save")
```

3.2.2 Saving

See Section 2.3 above for more details.

Appendix

A Generate random landscapes (randomLandscapes)

```
> pkgs <- list("SpaDES", "raster", "RColorBrewer")

> loadPackages(pkgs)

> rm(pkgs)

> doEvent.randomLandscapes <- function(sim, eventTime,
+     eventType, debug = FALSE) {
+     if (eventType == "init") {
+         depends <- NULL
+         if (reloadModuleLater(sim, depends)) {
+             sim <- scheduleEvent(sim, simCurrentTime(sim), "randomLandscapes",
+                 "init")
+         }
+         else {
+             sim <- randomLandscapesInit(sim)
+         }
+         sim <- scheduleEvent(sim, simParams(sim)$randomLandscapes$.plotInitialTime,
+             "randomLandscapes", "plot")
+         sim <- scheduleEvent(sim, simParams(sim)$randomLandscapes$.saveInitialTime,
+             "randomLandscapes", "save")
+     }
+     else if (eventType == "plot") {
+         Plot(get(simGlobals(sim)$.stackName, envir = .GlobalEnv))
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$randomLandscapes$.plotInterval,
+             "randomLandscapes", "plot")
+     }
+     else if (eventType == "save") {
+         saveFiles(sim)
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$randomLandscapes$.saveInterval,
+             "randomLandscapes", "save")
+     }
+     else {
+         warning(paste("Undefined event type: '", simEvents(sim)[1,
+             "eventType", with = FALSE], "' in module '", simEvents(sim)[1,
+             "moduleName", with = FALSE], "'", sep = ""))
+     }
+     return(invisible(sim))
+ }

> randomLandscapesInit <- function(sim) {
+     if (is.null(simParams(sim)$randomLandscapes$inRAM)) {
+         inMemory <- FALSE
+     }
+     else {
+         inMemory <- simParams(sim)$randomLandscapes$inRAM
+     }
+     nx <- simParams(sim)$randomLandscapes$nx
+     ny <- simParams(sim)$randomLandscapes$ny
+     template <- raster(nrows = ny, ncols = nx, xmn = -nx/2, xmx = nx/2,
```

```
+         ymn = -ny/2, ymx = ny/2)
+     speedup <- max(1, nx/500)
+     DEM <- round(GaussMap(template, scale = 300, var = 0.03,
+         speedup = speedup, inMemory = inMemory), 1) * 1000
+     forestAge <- round(GaussMap(template, scale = 10, var = 0.1,
+         speedup = speedup, inMemory = inMemory), 1) * 20
+     forestCover <- round(GaussMap(template, scale = 50, var = 1,
+         speedup = speedup, inMemory = inMemory), 2) * 10
+     percentPine <- round(GaussMap(template, scale = 50, var = 1,
+         speedup = speedup, inMemory = inMemory), 1)
+     forestAge <- forestAge/maxValue(forestAge) * 100
+     percentPine <- percentPine/maxValue(percentPine) * 100
+     habitatQuality <- (DEM + 10 + (forestAge + 2.5) * 10)/100
+     habitatQuality <- habitatQuality/maxValue(habitatQuality)
+     mapStack <- stack(DEM, forestAge, forestCover, habitatQuality,
+         percentPine)
+     names(mapStack) <- c("DEM", "forestAge", "forestCover", "habitatQuality",
+         "percentPine")
+     name(mapStack) <- simGlobals(sim)$.stackName
+     setColors(mapStack) <- list(DEM = terrain.colors(100), forestAge = brewer.pal(9,
+         "BuGn"), forestCover = brewer.pal(8, "BrBG"), habitatQuality = brewer.pal(8,
+         "Spectral"), percentPine = brewer.pal(9, "Greens"))
+     assign(simGlobals(sim)$.stackName, mapStack, envir = .GlobalEnv)
+     simModulesLoaded(sim) <- append(simModulesLoaded(sim), "randomLandscapes")
+     return(invisible(sim))
+ }
```

B Simulate fire spread on a landscape (fireSpread)

```
> pkgs <- list("SpaDES", "raster", "RColorBrewer")

> loadPackages(pkgs)

> rm(pkgs)

> doEvent.fireSpread <- function(sim, eventTime, eventType,
+     debug = FALSE) {
+     if (eventType == "init") {
+         depends <- NULL
+         checkObject(simGlobals(sim)$.stackName, layer = "habitatQuality")
+         if (!exists(simGlobals(sim)$burnStats, envir = .GlobalEnv)) {
+             assign(simGlobals(sim)$burnStats, numeric(), envir = .GlobalEnv)
+         }
+         else {
+             npix <- get(simGlobals(sim)$burnStats, envir = .GlobalEnv)
+             stopifnot("numeric" %in% is(npix), "vector" %in%
+                 is(npix))
+             if (length(npix) > 0) {
+                 message(paste0("Object `", simGlobals(sim)$burnStats,
+                   "` already exists and will be overwritten."))
+             }
+         }
+         if (reloadModuleLater(sim, depends)) {
+             sim <- scheduleEvent(sim, simCurrentTime(sim), "fireSpread",
+                 "init")
+         }
+         else {
+             sim <- fireSpreadInit(sim)
+             sim <- scheduleEvent(sim, simParams(sim)$fireSpread$startTime,
+                 "fireSpread", "burn")
+             sim <- scheduleEvent(sim, simParams(sim)$fireSpread$.saveInterval,
+                 "fireSpread", "save")
+             sim <- scheduleEvent(sim, simParams(sim)$fireSpread$.plotInitialTime,
+                 "fireSpread", "plot.init")
+         }
+     }
+     else if (eventType == "burn") {
+         sim <- fireSpreadBurn(sim)
+         sim <- scheduleEvent(sim, simCurrentTime(sim), "fireSpread",
+             "stats")
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$fireSpread$returnInterval,
+             "fireSpread", "burn")
+     }
+     else if (eventType == "stats") {
+         sim <- fireSpreadStats(sim)
+     }
+     else if (eventType == "plot.init") {
+         maps <- get(simGlobals(sim)$.stackName, envir = .GlobalEnv)
+         setColors(maps) <- list(DEM = terrain.colors(100), forestAge = brewer.pal(9,
+             "BuGn"), forestCover = brewer.pal(8, "BrBG"), habitatQuality = brewer.pal(8,
+             "Spectral"), percentPine = brewer.pal(9, "Greens"),
```

```
+             Fires = c("#FFFFFF", rev(heat.colors(9))))
+         Plot(maps)
+         assign(simGlobals(sim)$.stackName, maps, envir = .GlobalEnv)
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$fireSpread$.plotInterval,
+             "fireSpread", "plot")
+     }
+     else if (eventType == "plot") {
+         Plot(get(simGlobals(sim)$.stackName)$Fires, add = TRUE)
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$fireSpread$.plotInterval,
+             "fireSpread", "plot")
+     }
+     else if (eventType == "save") {
+         saveFiles(sim)
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$fireSpread$.saveInterval,
+             "fireSpread", "save")
+     }
+     else {
+         warning(paste("Undefined event type: '", simEvents(sim)[1,
+             "eventType", with = FALSE], "' in module '", simEvents(sim)[1,
+             "moduleName", with = FALSE], "'", sep = ""))
+     }
+     return(invisible(sim))
+ }

> fireSpreadInit <- function(sim) {
+     landscapes <- get(simGlobals(sim)$.stackName, envir = .GlobalEnv)
+     Fires <- raster(extent(landscapes), ncol = ncol(landscapes),
+         nrow = nrow(landscapes), vals = 0)
+     names(Fires) <- "Fires"
+     setColors(Fires, 10) <- c("#FFFFFF", rev(heat.colors(9)))
+     Fires <- setValues(Fires, 0)
+     assign(simGlobals(sim)$.stackName, addLayer(landscapes, Fires),
+         envir = .GlobalEnv)
+     simModulesLoaded(sim) <- append(simModulesLoaded(sim), "fireSpread")
+     return(invisible(sim))
+ }

> fireSpreadBurn <- function(sim) {
+     landscapes <- get(simGlobals(sim)$.stackName, envir = .GlobalEnv)
+     Fires <- spread(landscapes[[1]], loci = as.integer(sample(1:ncell(landscapes),
+         simParams(sim)$fireSpread$nFires)), spreadProb = simParams(sim)$fireSpread$spreadprob,
+         persistance = simParams(sim)$fireSpread$persistprob,
+         mask = NULL, maxSize = 1e+08, directions = 8, iterations = simParams(sim)$fireSpread$its,
+         plot.it = FALSE, mapID = TRUE)
+     names(Fires) <- "Fires"
+     setColors(Fires, 10) <- c("#FFFFFF", rev(heat.colors(9)))
+     landscapes$Fires <- Fires
+     assign(simGlobals(sim)$.stackName, landscapes, envir = .GlobalEnv)
+     return(invisible(sim))
+ }

> fireSpreadStats <- function(sim) {
+     npix <- get(simGlobals(sim)$burnStats, envir = .GlobalEnv)
+     landscapes <- get(simGlobals(sim)$.stackName, envir = .GlobalEnv)
```

```
+     assign("nPixelsBurned", c(npix, length(which(values(landscapes$Fires) >
+         0))), envir = .GlobalEnv)
+     return(invisible(sim))
+ }

> fireSpreadStats <- function(sim) {
+     npix <- get(simGlobals(sim)$burnStats, envir = .GlobalEnv)
+     landscapes <- get(simGlobals(sim)$.stackName, envir = .GlobalEnv)
+     assign("nPixelsBurned", c(npix, length(which(values(landscapes$Fires) >
+         0))), envir = .GlobalEnv)
+     return(sim)
+ }
```

C Agent based model of caribou movement (caribouMovement)

```
> pkgs <- list("SpaDES", "grid", "raster", "sp")

> loadPackages(pkgs)

> rm(pkgs)

> doEvent.caribouMovement <- function(sim, eventTime,
+     eventType, debug = FALSE) {
+     if (eventType == "init") {
+         depends <- NULL
+         checkObject(simGlobals(sim)$.stackName, layer = "habitatQuality")
+         if (reloadModuleLater(sim, depends)) {
+             sim <- scheduleEvent(sim, simCurrentTime(sim), "caribouMovement",
+                 "init")
+         }
+         else {
+             sim <- caribouMovementInit(sim)
+             sim <- scheduleEvent(sim, 1, "caribouMovement", "move")
+             sim <- scheduleEvent(sim, simParams(sim)$caribouMovement$.plotInitialTime,
+                 "caribouMovement", "plot.init")
+             sim <- scheduleEvent(sim, simParams(sim)$caribouMovement$.saveInitialTime,
+                 "caribouMovement", "save")
+         }
+     }
+     else if (eventType == "move") {
+         sim <- caribouMovementMove(sim)
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$caribouMovement$moveInterval,
+             "caribouMovement", "move")
+     }
+     else if (eventType == "plot.init") {
+         Plot(caribou, addTo = "forestAge", add = TRUE, size = 1,
+             pch = 19, gp = gpar(cex = 0.01))
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$caribouMovement$.plotInterval,
+             "caribouMovement", "plot")
+     }
+     else if (eventType == "plot") {
+         Plot(caribou, addTo = "forestAge", add = TRUE, pch = 19,
+             size = 1, gp = gpar(cex = 0.01))
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$caribouMovement$.plotInterval,
+             "caribouMovement", "plot")
+     }
+     else if (eventType == "save") {
+         saveFiles(sim)
+         sim <- scheduleEvent(sim, simCurrentTime(sim) + simParams(sim)$caribouMovement$.saveInterval,
+             "caribouMovement", "save")
+     }
+     else {
+         warning(paste("Undefined event type: '", simEvents(sim)[1,
+             "eventType", with = FALSE], "' in module '", simEvents(sim)[1,
+             "moduleName", with = FALSE], "'", sep = ""))
+     }
+     return(invisible(sim))
```

```
+ }

> caribouMovementInit <- function(sim) {
+     landscape <- get(simGlobals(sim)$.stackName, envir = .GlobalEnv)
+     yrange <- c(ymin(landscape), ymax(landscape))
+     xrange <- c(xmin(landscape), xmax(landscape))
+     N <- simParams(sim)$caribouMovement$N
+     IDs <- as.character(1:N)
+     sex <- sample(c("female", "male"), N, replace = TRUE)
+     age <- round(rnorm(N, mean = 8, sd = 3))
+     x1 <- rep(0, N)
+     y1 <- rep(0, N)
+     starts <- cbind(x = runif(N, xrange[1], xrange[2]), y = runif(N,
+         yrange[1], yrange[2]))
+     caribou <<- SpatialPointsDataFrameNamed(coords = starts,
+         data = data.frame(x1, y1, sex, age), name = "caribou")
+     row.names(caribou) <<- IDs
+     return(invisible(sim))
+ }

> caribouMovementMove <- function(sim) {
+     landscape <- get(simGlobals(sim)$.stackName, envir = .GlobalEnv)
+     caribou <<- crop(caribou, landscape)
+     if (length(caribou) == 0)
+         stop("All agents are off map")
+     ex <- landscape[["habitatQuality"]][caribou]
+     sl <- 0.25/ex
+     ln <- rlnorm(length(ex), sl, 0.02)
+     sd <- 30
+     caribou <<- move("crw", caribou, stepLength = ln, stddev = sd,
+         lonlat = FALSE)
+     return(invisible(sim))
+ }
```