Agent-based model of rhino poaching

Model description

Jacob van der Ploeg (S2970783)
vanderploeg.jacob@gmail.com
University of Groningen, Master of Sociology (criminology)

Internship Netherlands Institute for the Study of Crime and Law Enforcement
Supervisors: Michael Mäs (RUG) and Andrew Lemieux (NSCR)
Model development by Nick van Doormaal (NSCR)

Amsterdam, May 2017

Table of contents

Introduction: rhino poaching agent-based model description

The rhino poaching model is an agent-based model simulating the dynamic interaction between three agents involved in a particular form of wildlife crime: active poaching. The included agents are rhinos, poachers and rangers. Each individual agent operates on specific decision rules set for each agent group, based on their environment which includes and is affected by other agents. The interaction between agents in this crime process is described as the *triple-foraging process* by Lemieux (2014). All agents are directed by the resource they are "foraging" for: animals look for food and water, poachers look for animals and rangers look for poachers. This model is meant to help understand the variables affecting the outcomes of this process and identify environmental characteristics and behavioral rules of relevance. It can be used to find optimal ranger strategies to keep alive as many rhinos and arrest as many poachers as possible, under varying environmental circumstances and dealing with various poaching strategies.

In this document the model and its guiding principles will be explained to help understand the model outcomes and identify new lines of inquiry that can be studied using this tool. Relevant code can be found in the model's code tab via the *procedures* drop-down menu. In some places it will be added in this document. The interface that is shown when first opening the model and the use thereof will first be described, followed by the setup of the world and initialization of its agents and finally the decision rules these agents operate on. At the end of the document a glossary is added, giving a brief description of a number of terms in the model.

1 - Interface

Relevant code: `setup-park, setup-humans`

This paragraph gives a first basic introduction to the model as seen when first opening it. Details of procedures and variables are discussed in paragraphs on the world and agents specifically. When opening the model the interface shows an empty (black) world in which the simulations will be presented (figure 1). The grey buttons in the top left are used to set up the world by creating its

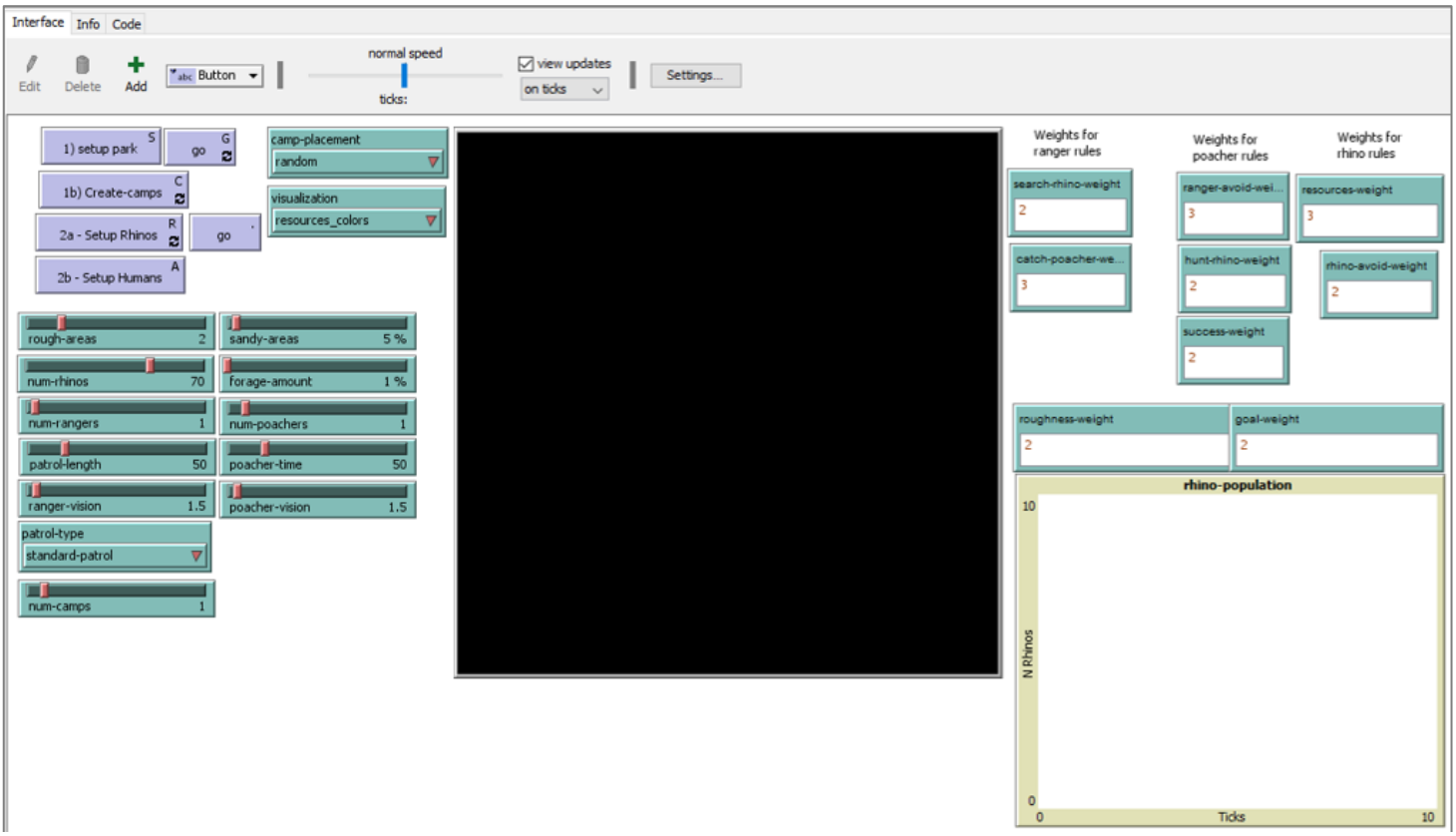

Figure 1: Model interface before setup.

variables. Each of these buttons calls on a procedure in the code to create the world and also run the simulation (the symbols in the upper right corners of these buttons are keyboard action keys). The green sliders, drop-down menu's and entry boxes allow for setting variable values and determining patrol type, before setting up the world. The graph to the right plots the count of surviving rhinos. If relevant as an outcome measure a poacher count can be added: right click the graph > add pen > pen update commands: *plot count poachers*.

The first step in using the interface is to determine the values for each of the variables in the green interface items (default values can be maintained for understanding the model and going through this description). The sliders determine characteristics of the world and agents, whereas the drop-down menu's let you choose how to run and visualize the simulation. Values in the entry boxes are used in the equations for each agent's decision rules, allowing the user to alter relative importance of variables for each of the agents. After the variables have been set the *1) setup park* button is used to create the environment (called *park* or *reserve* in this document). This button calls on procedures that set variables which are the same for each agent (globals). These include resource regrowth rate (*sprout-delay-time,* which is given a base rate of 100 ticks), maximum roughness of a patch (5), rhino sign decay rates (500) and the *rhino-weight* which is the sum of variables in the rhino's decision rule (determined in the entry boxes). Furthermore it calls on the procedures that create the park resources, set its roughness, release the rhinos and set rhino home ranges.

At this stage, when selecting the *resources_colors* option from the *visualization* menu, you see a world with varying shades of green that represent the resource levels in each patch. Brown patches represent the sandy areas. When having selected the *rhino_home-ranges* visualization you see the same world with colored areas around each rhino agent. These represent their home-ranges, which will expand as they start moving (as shown in figure 2). Finally selecting the *rough_areas* visualization shows you red circles that fade out from the center, representing the number of rough areas determined in the slider. The black patches surrounding the park represent a hard border which cannot be crossed by the animal agents and do not contain resources. Note: when the simulation is running you can alternate between these visualizations. If you want to change the visualization at this stage of the set-up you need to press *setup park* again after making your changes.

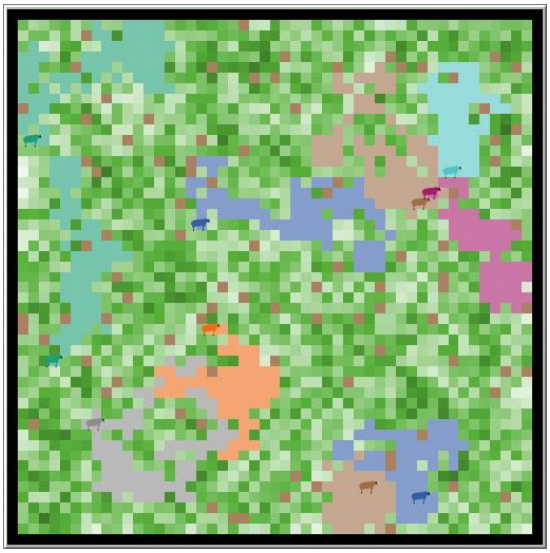

Figure 2: View of the park showing resources (shades of green), borders (black), rhinos (agents in various colors) and home ranges (in color of corresponding rhino).

The following step depends on a number of choices. If you want to manually place the ranger camps select *we-decide* from the *camp-placement* menu. Now click *1b) Create-camps* (or press C) and select a patch where you want to place a camp. Repeat this for every camp. The default setting allows you to skip this step as it places the camps randomly with a certain buffer to avoid being placed too close to one another: select *random* in *camp-placement* (button *1b) create-camps* can then be ignored). For running NetLogo experiments this is more userfriendly than manually selecting camp locations. When running experiments it may be preferrable to first run the simulation with only rhinos in the park. This allows for a learning period in which the rhinos can establish their home range. To do so click *2a – Setup Rhinos* (or press R). The rhinos will now start moving for 1000 ticks. Alternatively you may have all agents start together. To do so ignore *2a – Setup Rhinos* and proceed to *2b – Setup Humans (or press A)*. This button calls on the procedure to create camps as well as enter rangers and poachers

(when camp-placement is set to random). Camps are shown as red houses with a resource-empty buffer around them which prevents rhinos swarming to them (figure 3), ranger agents are green and poacher agents – which enter at the border – are blue. Finally you may now press *go* to set the world in motion. The *go* button with the circular arrows runs the simulation continuously untill stopped by the user, a lack of rhinos, a lack of poachers or the passing of 10000 ticks. The other *go* button moves the world one tick at a time, better allowing users to see changes.

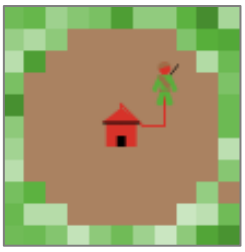

Figure 3: Ranger camp

So what makes this world and its agents tick? The following chapters will describe the world's environment with all its variables, the agents and what decision rules and equations are used for their movement.

2 - Set-up

2.1. Park

Relevant code: `setup-park` (button *1 setup park*), calling `create-park`, `create-rough-areas`, `release-rhinos`, `setup-home-range` and corresponding sub-procedures.

When setting up the world the procedure `create-park` is called to set the contents and characteristics of each patch. Some of these may be seen in figure 4 which shows the information you get when right-clicking a patch and selecting *inspect*. It is here that the global variables *sprout-delay-time*, *max-roughness* and *rhino-activity-decay* are set at 100, 5 and 500 ticks respectively. Patches are designated *inside* the reserve when they have 8 neighboring patches, thereby excluding only the border patches which are made black and are given a resource value of 0. All patches start out belonging to nobody, which may change when at a later stage included in a rhino territory as it moves through the patch. At this stage all patches are given a roughness value of 1 in a range of 1-5, with 1 being low roughness and therefor easiest (faster) to move through (see agent movement). Each patch is given a random amount of resources represented by a value between 0 and 1. This variable value is diffused to its neighbors to create smoother, more natural gradients (`diffuse resources 0.4`): as such each patch gets 1/8 of 40% of its resources from each of its neighbors. The patches are given a color along the green shade with light green representing low resources and darker green representing higher resources (procedure `color-resources`). Furthermore each patch is given a countdown made up of the *sprout-delay-time* plus a random number of ticks between 0 and 100. This makes for varying regrowth of resources across the park. A percentage of patches is made sandy (brown), as determined by the slider in the interface, also having a resource value of 0.

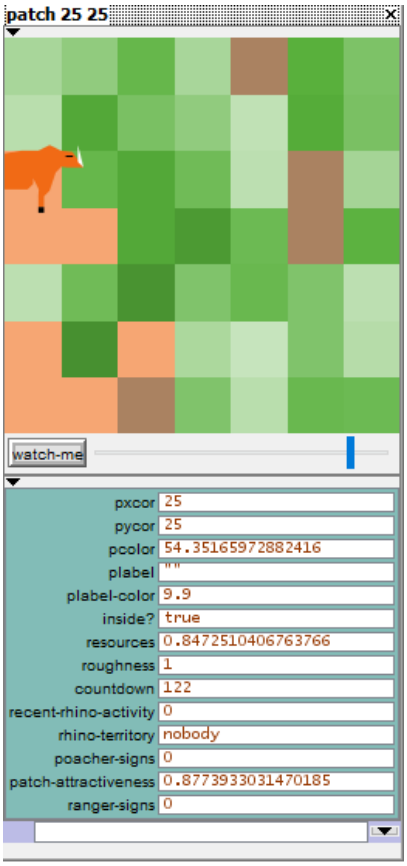

Figure 4: Patch information (center patch)

With this setup the `create-rough-areas` procedure is also called. The number of rough areas, also specified with a slider, is the number of patches that are given the maximum roughness value of 5. The patches surrounding these max-value patches are subsequently selected and given a roughness value within the 1 - 5 range, based on their proximity to the max-value patch. This fades out the roughness in a circle with the highest value patch in the center:

```
ask n-of rough-areas patches with [inside? = TRUE] [
       set roughness max-roughness
ask patches in-radius ( max-roughness - 1 ) [
       set roughness round (max-roughness - distance myself + roughness)
…]
```

This roughness variable is later used to determine movement speed through, and attractiveness of patches. When the *color-rough-areas* visualization is selected, patches with roughness greater than 1 are colored red, increasing in darkness as roughness gets higher.

The remaining patch-owned items that can be seen in figure 4 are (partly) set by agents and will be discussed when their setup and operating is described. Park setup also releases rhinos into the world (`release-rhinos`). The first step sets the *rhino-weight* value which is later used for rhino movement as part of the *patch-attractiveness*. It consists of the sum of rhino avoidance, roughness and resource weights that are entered by the user in the interface.

Eq. 1) rhino-weight = rhino-avoid-weight + roughness-weight + resources-weight

Then to enter the rhinos a number of patches equal to the number of rhinos specified with the slider sprouts 1 rhino agent. No other agent may be on a patch in a radius of 5 for a rhino to sprout somewhere. It is here in the code that some of the rhino-owned variables are created: *rhino-memories* are patchset consisting of a list of the 3 previously visited patches (at this stage of the setup only the current location). They are also given a *goal*, which will later be used to direct their movement: this is set to the current patch. Finally the `release-rhinos` procedure calls on the `spread-rhinos` procedure to further establish their territoriality. Rhinos that need further spreading are those that have an agent in a radius of 10 patches (note that at this stage these agents only include other rhinos). If there are patches available that have no other rhinos in a 10 patch radius, these rhinos move there. If no such patches are available the spreading rhinos move to any unoccupied patch within the reserve:

```
to spread-rhinos
let spreading-rhinos rhinos with [any? other turtles in-radius 10]
if any? spreading-rhinos [
  ask one-of spreading-rhinos [
    let available-patches patches with [inside? = TRUE and not any? other
    turtles in-radius 10]
   ifelse any? available-patches [
      move-to one-of available-patches
      spread-rhinos]
    [ move-to one-of patches with [inside? = TRUE and not any? other turtles-
    here ] ]
]]
```

The final remaining procedure being called by `setup-park` also deals with rhinos. `Setup-home-range` determines in part what happens to patches as rhinos enter the world. All patches within a radius of 1.5 patches around a rhino are set as its *rhino-territory*, and will belong to this agent. As rhinos start moving this home range will expand and meet with other home ranges and rhinos, party directing rhino movement (discussed in the paragraph on movement rules for the agents). If a random value between 0 and 1 is below 0.5 a rhino visited patch is given signs of *recent-rhino-activity*. The *rhino-activity-decay* rate now is set at the previously determined minimum of 500 ticks plus an additional random amount between 0 and 500 ticks. This creates varying (amounts of) signs.

You may notice another patch-owned variable in the menu of figure 4 is called *patch-attractiveness*. As this is in part determined by agents that have not been introduced yet it will be discussed at a later stage. As the result of a number of variables for each specific agent type and used to create weighted probability of selection of that patch as the next movement goal it is crucial in agent movement.

2.2. Human agents

Relevant code: `setup-humans` (button *2b setup humans*), calling `build-camps`, `distribute-camps`, `release-rangers` and `release-poachers`

The human agents enter the world through a separate procedure to allow for running the rhino only learning period: `setup-humans`. Human agents are poachers and rangers who, like rhinos, leave signs. The most persistent of these signs is a carcass that is left after a poacher is successful in killing a rhino. Its decay rate is set to 10000 ticks, equal to the maximum number of ticks in the simulation and as such permanently visible. Human activity signs are left in a similar way to rhino signs with a decay rate of 200 ticks. Similar to rhinos the *poacher-* and *ranger-weights* used for movement decision making as part of *patch-attractiveness* are set here, consisting of user entered weights for the relevant variables. (The + 1 in equation 3 is to make ranger patrols avoid one another).

Eq. 2) $poacher\text{-}weight = (ranger\text{-}avoid\text{-}weight * 2) + hunt\text{-}rhino\text{-}weight + roughness\text{-}weight + resources\text{-}weight$

Eq. 3) $ranger\text{-}weight = 1 + search\text{-}rhino\text{-}weight + roughness\text{-}weight + catch\text{-}poacher\text{-}weight$

The first procedure called by `setup-humans` builds the camps from which the rangers operate, shown as houses. When placement is random a number of patches equal to the number of camps specified with the slider will sprout 1 camp. When placement is fixed the camps are placed on a number of pre-specified patches with a small degree of randomness. Finally, manually placing the camps uses the procedure `click-camps` which is called by button *1b create camps* and places a camp on mouse-clicked patches. After placing the camps `distribute-camps` is called to avoid cluttering which would negate some effects of additional camps. Code similar to the spreading of rhinos is used to let camps that are within a radius of 10 patches of another camp move to a patch without camps its radius. If no such option is available the camp is placed on a random patch within the park. This procedure also asks camps to set resources in a radius of 4.5 patches around them to 0, thereby avoiding cluttering of rhinos around what may become relatively safe havens and unnatural attraction of rhinos to human hotspots.

Rangers, as green human figures, now enter the world. The number of rangers is specified by the slider, and for each of them one of the camps is asked to hatch one ranger agent. These rangers own a *patrol-memory* of previously visited patches (set to no-patches), *patrol-time* recording the duration of current patrol, memory of *risky-areas* and *safe-areas* where poacher signs have and have not been detected by this ranger (both set to no-patches at this stage), memory of *rhino-sightings* (set to no-patches) and a waiting period between patrols known as *off-duty-time* and defined as *(patrol-length * 2) + random 100* (a random value between 0 and 100). Like rhinos, they too have a *goal* which is to be used for movement later (currently the current patch `patch-here`). The *state* of rangers, which tells you whether they are on patrol or off-duty, are set to *patrolling* upon *release*. These states allow for calling corresponding movement procedures (i.e. `if state is patrolling` …).

Finally `setup-humans` calls on the procedure to `release-poachers`. A number of patches outside the reserve (black border patches) equal to the number of poachers that has been set by the slider sprouts 1 poacher, shown as a blue human figure. Like rangers and rhinos these poachers own a number of attributes. The *time* measures how long they have been inside the reserve (at this stage set to 0 ticks), a *laylow* period describes the time they wait in between hunts and is defined as *(poacher-time * 2) + random 100*. Furthermore, they have memories storing their good hunting grounds (*good-sites*), *success-sites, failure-sites* and previous hunting trips named *poacher-memory* (all set to no-patches at this stage). Like the other agents they too have a *goal* (`current-patch`), and a *state* similar to rangers to describe if they are in the process of hunting or laying low. At this entry stage, the poacher *states* are set to *journey-to-crime*. These states again correspond to relevant procedures for agent actions.

3 - Agent movement
Relevant code: `go` (*go* and *setup rhinos* buttons), calling `rhino-goal-check`, `update-rhino-memory`, `rhino-foraging-and-home-range`, `poacher-goal-check`, `poacher-laylow`, `ranger-goal-check`, `ranger-end-patrol`, `update-resources`, `update-home-range`, `update-visualization`, `update-activity-signs` and (many) corresponding sub-procedures.

As described in the interface paragraph it is possible to run the world with rhinos only (button 2a). The rhinos and patches are then called to run the same procedures as when simply running all agents at once with the *go* button (despite their names the *setup rhino* buttons and procedure do not set up this agent type in the same way as the poachers and rangers are set up in the so called procedures, as rhinos are initialized in the `create-park` procedure. *Setup rhino* refers here to the learning period that allows rhinos to establish themselves). As such, the movement will be described in order of the `go` procedure, noting that the `setup-rhino` procedure and button call the same rhino and patch procedures for a duration of 1000 ticks (`rhino-goal-check`, `update-rhino-memory`, `rhino-foraging-and-home-range`, `update-resources`, `update-home-range`, `update-visualization` and `update-activity-signs`).

The first thing established in this procedure that starts the simulation is when said simulation should end. If no rhinos or poachers remain, or when 10000 ticks have passed the run halts. The 10000 ticks can be thought of as representing a year when related to the time available for rangers to patrol and poachers to hunt.

To set the agents in motion the first thing that is determined for all types is the movement speed:
Each agent is given a waiting time related to the roughness of the current patch (ticks-to-stay-on-patch = *roughness - 1*). All agents own another variable called *ticks-wait*, which is set equal to `ticks-to-stay-on-patch` when roughness is greater than 1 and counts down - 1 every tick. As such an agent on a patch with roughness value 5 waits 4 ticks before moving on to the next patch (counted down by *ticks-wait*), whereas an agent on a patch with roughness value 1 moves the very next tick. This is coded in each agent's `goal-check` procedure, called by `go`, given here for the rhinos:

```
to-report ticks-to-stay-on-patch [p]
  report roughness - 1
end

to rhino-goal-check
ifelse ticks-wait > 0 [
  set ticks-wait ticks-wait - 1 ]
…
if ( roughness > 1)
  [ set ticks-wait ticks-to-stay-on-patch patch-here ]
]
end
```

3.1. Rhino behavior
The `goal-check` procedures then further determine the agent's movement following more agent type specific procedures. This paragraph first describes the rhinos, followed by poachers and rangers.
If the rhino is on its current goal patch the `rhino-decision-making` procedure is called, in which the next goal is determined for the rhino to move to. Remember the goal at initialization was set to `patch-here` suggesting the goal has been reached. The first thing that happens is thus the selection of a new goal. Alternatively, when the rhino is not on its goal patch it follows that it is moving towards a previously set goal and the `rhino-movement` procedure runs.

3.1.1. Rhino-decision-making
To determine where to go (set a *goal*) each rhino looks at his *surroundings*, which are defined as all patches within a 2.5 radius that are within the park, outside its current home range and not recently

visited (i.e. included in its *rhino-memory* consisting of the last 3 patches visited). The *patch-attractiveness* of the patches in these *surroundings* is determined using the user-entered weights of the variables; negatively valuing roughness and positively valuing resources.

```
ask surroundings [
  let estimated-roughness ((max-roughness - roughness) / max-roughness) *
       roughness-weight
  let estimated-resources resources * resources-weight
…
```

Also included in the *patch-attractiveness* is the *rhino-dist*(ance) which consists of the user-entered *rhino-avoid-weight* multiplied by the distance of the nearest rhino.

```
to-report rhino-dist
  ifelse any? other rhinos [
    let max-rhino-dist distance min-one-of other rhinos [distance myself]
    ifelse max-rhino-dist = 0 [
      report 0 ]
    [ report (distance min-one-of other rhinos [distance myself] / max-rhino-
    dist) * rhino-avoid-weight ]
  ]
  [ report 0 ]
end
```

The patch attractiveness then consists of the sum of these values divided by the rhino-weight from equation 1 (*rhino-avoid-weight + roughness-weight + resources-weight*).

```
ask surroundings [
       …
       set patch-attractiveness (estimated-resources + estimated-roughness +
       rhino-dist) / rhino-weight
  ]
```

Attractiveness of patches belonging to other rhinos is minimized by:

```
ask surroundings with [rhino-territory != myself and rhino-territory
!= nobody] [
    set patch-attractiveness patch-attractiveness / 10]
```

Which does not exclude these patches as options. It is possible for rhinos to overtake another rhino's home range. Finally, the *goal* can now be set. To do so a NetLogo extension is used, which was added in the very first line of the code: `extensions [rnd]`. Now using `rnd:weighted-one-of` a random patch is selected from the *surroundings* agentset, with a selection probability based on the *patch-attractiveness*. If the resulting patch is a neighboring patch, the rhino will move there. If instead the resulting patch is further away in the 2.5 radius, the rhino will move towards it using the `rhino-movement` procedure (introduced at the start of the rhino movement paragraph as the procedure used when rhinos are not on their *goal* patch).

3.1.2. Rhino-movement

For moving towards the *goal,* similar procedures are used as when moving to the goal. First, the *surroundings* are defined as all neighboring patches within the park. Then the same formulas are used to determine *patch-attractiveness*, now also incorporating distance to the *goal* (*goal-dist*) and the user-entered weight of said *goal*. This again determines *patch-attractiveness*, minimizing other rhino territories, and selecting a patch using weighted probabilities. The inclusion of the *goal* and the weight thereof make for a greater likelihood of moving towards the set goal than elsewhere:

```
to rhino-movement
  let surroundings neighbors with [inside? = TRUE]
  let max-goal-dist distance rhino-goal
  ask surroundings [
    let goal-dist (max-goal-dist - distance [rhino-goal] of myself) * goal-
       weight
    let estimated-roughness ((max-roughness - roughness) / max-roughness) *
       roughness-weight
    let estimated-resources resources * resources-weight
    set patch-attractiveness (goal-dist + estimated-resources + estimated-
       roughness + rhino-dist) / (rhino-weight + goal-weight)
    if patch-attractiveness < 0 [
      set patch-attractiveness 0 ]
  ]
  ask surroundings with [rhino-territory != myself and rhino-territory
       != nobody] [
    set patch-attractiveness patch-attractiveness / 2 ]

  move-to rnd:weighted-one-of surroundings [patch-attractiveness]
end
```

3.1.3. Other rhino behavior

Continuing the `go` procedure, after the extensive `rhino-goal-check` and `rhino-movement` to make rhinos move, calls `update-rhino-memory` and `rhino-foraging-and-home-range`. The former makes each rhino save its current patch in its memory. When three patches have been saved, a fourth results in the first being removed, whereby the *rhino-memory* consists of the last 3 patches visited. Its use has already been described in paragraph 3.1.1. on `rhino-decision-making` where it makes sure rhinos do not end up in an infinite loop moving back and forth between the same two patches; as it avoids patches stored in its memory. The latter of these two procedures, `rhino-foraging-and-home-range`, describes the effect of the rhinos on their environment: the patches. Resources on each rhino's current patch diminish by a percentage set by the slider (equation: *set resources (resources * ((100 - forage-amount) / 100))*) to represent grazing and a countdown is set to match the `sprout-delay-time` of 100 ticks before resources can grow back. Furthermore the current patch is added to the rhino's home range. These home-ranges are given the color of the corresponding rhino and can be monitored in the home-range visualization. Another effect the rhinos have on the patches is the previously described leaving of signs (paragraph 2.1.).

3.2. Poacher behavior

To run the simulation the `go` procedure naturally also sets the poachers in motion. Similar to rhinos this is directed by setting a goal, checking fulfilment thereof and moving to or towards it (`poacher-goal-check` and `poacher-movement`). Poacher movement it tracked by entering `pen-down`. If poachers do not need to "wait" as a consequence of rough terrain, when they have time left to hunt and when they have yet to obtain a rhino horn (successful kill) their state is set to `journey-to-crime` and the procedure `poacher-journey-to-crime` is called. If time has run out (set by slider) or the hunt was successful, the state is set to `journey-after-crime` and the procedure `poacher-journey-after-crime` is called to make poachers leave the park via the shortest route. The `poacher-goal-check` also contains the code for leaving signs at a probability of 0.5, and keeps the time for these agents (+1 every tick, which when matching the available *poacher-time* results in park departure).

3.2.1. Poacher-journey-to-crime

Poachers set as a target one of the rhinos that is within a radius specified by the `poacher-vision` slider. If such a target is within 1.5 radius of the poacher, he moves there and kills the rhino, as per the `poacher-kills-rhino` procedure. The poacher is then on his goal patch and adds it to his patchset of *success-site*s. A permanent poacher sign is left in the form of a carcass. If the poacher is on his goal patch the `poacher-decision-making` procedure is called, which is similar to its rhino counterpart in that it sets a new poacher goal and makes the poacher move towards it. Again, the alternative is that

the poacher is still in the process of moving towards his previously set goal and `poacher-movement` is called. The patches visited during the journey to the crime are stored in the *poacher-memory*, which is cleared upon a successful kill to only remember the kill site itself. Of these route patches those with > 0 rhino signs and 0 ranger signs are stored in the *good-sites* memory and those with ranger signs > 0 are stored as *failure-sites* (good- and failure sites overwrite one another so the most recent visit is leading).

3.2.1.1. Poacher-decision-making

In deciding where to go poachers first identify *nearby-rangers* as a turtle-set consisting of all rangers and ranger camps within the `poacher-vision` radius. When either rangers or camps are within this radius, the patches are added to the *failure-sites* patch-set and the poacher proceeds with `journey-after-crime` to leave the park. Like rhinos, poachers also set *surroundings* from which to select their next destination. These are all those patches within the `poacher-vision` that are inside the park. In these *surroundings* a number of parameters are created that then make up the *patch-attractiveness* for poachers, similar to the rhino procedure:

```
let surroundings patches in-radius poacher-vision with [inside? = TRUE]
ask surroundings [
  let rhino-signs (recent-rhino-activity / (rhino-activity-decay * 2)) *
       hunt-rhino-weight
  let nearby-ranger-signs ((human-signs-decay - ranger-signs) / human-signs-
       decay) * ranger-avoid-weight
  let estimated-roughness ((max-roughness - roughness) / max-roughness) *
       roughness-weight
  let estimated-resources resources * resources-weight
  set patch-attractiveness (ranger-dist + rhino-signs + nearby-ranger-signs +
       estimated-resources + estimated-roughness) / poacher-weight
  if patch-attractiveness < 0 [
      set patch-attractiveness 0 ]
]
```

Where `ranger-dist` reports the nearest ranger, incorporating the user entered `ranger-avoid-weight`.

```
to-report ranger-dist
  let nearby-rangers (turtle-set rangers camps)
  ifelse any? nearby-rangers [
    let max-ranger-dist distance min-one-of nearby-rangers [distance myself]
    ifelse max-ranger-dist != 0 [
      report (distance min-one-of nearby-rangers [distance myself] / max-
       ranger-dist) * ranger-avoid-weight ]
  [ report 0 ]
  ]
  [ report 0 ]
end
```

Again, the attractiveness of those patches that have been visited very recently is minimized.

```
ask surroundings with [member? self [poacher-memory] of myself] [
 set patch-attractiveness patch-attractiveness / 10
```

Like in rhino movement, a goal patch is then designated within these *surroundings* for the poachers to move to, with a weighted probability of selection based on *patch-attractiveness*. If such a goal patch is a neighboring patch the poacher will move there, if it is further away in the specified radius he will move towards it using `poacher-movement`.

3.2.1.2. Poacher-movement

The `poacher-movement` procedure also specifies as *nearby-rangers* those rangers and camps within the `poacher-vision`. In the same way that `rhino-movement` uses a similar approach as `rhino-decision-making`, `poacher-movement` uses the same setup as `poacher-journey-to-crime`. What is added to the *patch-attractiveness* is again the distance to the goal that has been set and the weight thereof:

```
let max-goal-dist distance poacher-goal

  ask neighbors with [inside? = TRUE] [
    let goal-dist (max-goal-dist - distance [poacher-goal] of myself) * goal-
      weight
    let rhino-signs (recent-rhino-activity / (rhino-activity-decay * 2)) *
      hunt-rhino-weight
    let nearby-ranger-signs ((human-signs-decay - ranger-signs) / human-
      signs-decay) * ranger-avoid-weight
    let estimated-roughness ((max-roughness - roughness) / max-roughness) *
      roughness-weight
    let estimated-resources resources * resources-weight
    set patch-attractiveness (goal-dist + estimated-resources + estimated-
      roughness + ranger-dist + rhino-signs + nearby-ranger-signs) /
      (poacher-weight + goal-weight)
    if patch-attractiveness < 0 [
      set patch-attractiveness 0 ]
  ]
  let surroundings neighbors with [inside? = TRUE and not member? self
      [poacher-memory] of myself]
  ifelse any? surroundings [
    move-to rnd:weighted-one-of surroundings [patch-attractiveness] ]
  [ move-to rnd:weighted-one-of neighbors with [inside? = TRUE] [patch-
attractiveness] ]
end
```

3.2.2. After the crime

`Poacher-journey-after-crime` describes what happens after a poacher has been successful in killing a rhino, when *poacher-time* has run out or when he wants to leave the park for spotting a ranger. In each of these cases the poacher *goal* is set as the nearest patch that is outside the reserve. If that is a neighboring patch the poacher will move there and if it is further away he will face towards it and move there one patch at a time. When the poacher state is set as *journey-after-crime* and the poacher has reached his goal patch he has successfully escaped the park. The *laylow* period now starts counting down. This is set as *(poacher-time * 2) + random 100*. Upon reaching 0 `poacher-plan-new-trip` is called.

To plan a new trip the *good-* and *success-sites* from poachers memories are designated as *good-options* (patch-set). If no such patches are available to a poacher he chooses a random patch along the border that is not part of his *failure-site* patch-set memory and starts from there. If *good-options* are available `rnd:weighted-one-of` is again used to make a weighted decision on where to plan the trip to. The *new-trip-weight* is made up of the user-entered *success-weight* + *hunt-rhino-weight* + *ranger-avoid-weight*, to allow prioritizing in this planning stage. *Patch-attractiveness* for the new trip is then made as follows:

```
ask good-options [
      let recent-rhino-kill (poacher-signs / carcass-decay) * success-weight
      let detected-rhino-signs (recent-rhino-activity / (rhino-activity-decay
            * 2)) * hunt-rhino-weight
      let detected-ranger-signs ((human-signs-decay - ranger-signs) / human-
            signs-decay) * ranger-avoid-weight

    set patch-attractiveness (recent-rhino-kill + detected-rhino-signs +
       detected-ranger-signs) / new-trip-weight

    if patch-attractiveness < 0 [
      set patch-attractiveness 0 ]
    ]
```

The *best-option* is the patch in the *good-options* set that scores highest on *patch-attractiveness*. To allow some explorative poacher behavior, instead of only revisiting sites from memory, this *best-option* is compared to a random value between 0 and 1. If this random value is below the *best-option's patch-attractiveness* the poacher will select a patch from the *good-options* within a 1.5 patch radius as movement goal, with a weighted probability based on their *patch-attractiveness* (`rnd:weighted-one-of`). The poacher then starts his trip at the border patch nearest to this goal. This simulates a poacher remembering a good site, with some degree of variability deciding whether to revisit that site and upon deciding so opting for the shortest route through the park. If on the other hand the random value is higher than the *best-option's patch-attractiveness* a random entry point and goal are selected.
Following this the poacher's state is again set to *journey-to-crime* to start the trip: his laylow period is reset (using *(poacher-time * 2) + random 100*), his time counter is reset to 0, his memory is cleared and his goal is set to the current patch.

3.3. Ranger behavior

The final agent that remains at this stage is the ranger, whose movement is also initiated by `go` and is very similar to that of poachers, again being tracked by the `pen-down` command and with roughness based speed set just like the other agents. Where poachers set rhinos as their target and move there to go for the kill, rangers set one of the poachers in the user-set *ranger-vision* and within the park as their target. If such a target is present, the `rangers-catch-poachers` procedure is called. Like poachers, rangers have a number of "states" that call their actions. If there is *patrol-time* left rangers can either be in the process of *patrolling* or *follow-up*. In case of the former the type of patrol is first determined as either *standard-patrol* or *fence-patrol* and the corresponding procedures are called. In case of the latter the `ranger-follow-up` procedure is called. Alternatively, when no time is left the ranger *state* is set to *back-to-camp* and the procedure `ranger-to-camp` is called: when the *patrol-time* counter exceeds the user-set *patrol-length* the ranger state changes to this *back-to-camp* state to call the procedure (*patrol-time* is monitored similar to poacher's time, by counting the ticks).

Rangers also have memories, storing visited patches with > 0 poacher signs and patches within his vision that have poacher agents on them as *risky-areas* (i.e. a ranger can see poachers and remember where without depending on visiting the exact patch and monitoring the signs). Those patches with 0 poacher signs are *safe-areas* (again overwriting one another when needed). Rangers also store their *rhino-sightings* of all rhinos on patches neighboring their own. Finally rangers leave signs at a 0.5 probability rate just like rhinos and poachers.

When a poacher is within the ranger's vision and inside the park the ranger moves to that location and catches the poacher (in NetLogo language the poacher "dies"). The patch is added to the *risky-areas*, the ranger goal is set to the current patch (goal achieved) and ranger state is set to *back-to-camp* to call the corresponding procedure. The ranger moves to the nearest camp, whereas `ranger-back-to-camp` also allows for leaving the park at the border if that is closer than the nearest camp (this is optional when ending a patrol because time has run out though not with someone in custody).

First however, rangers need to find poachers. When the ranger's state is *patrolling* and the *standard-patrol* is selected `ranger-patrol` is called. This is very much like the rhino and poacher decision making procedures. When the ranger has reached his goal patch a new goal is determined that keeps him in motion. If no goal is being determined it again follows that the agent is still moving towards its previously set goal. The goal is set by `ranger-decision-making` and the movement by `ranger-movement`.

3.3.1. Ranger-decision-making

Like rhinos and poachers, rangers look at their *surroundings* and select one patch from there as their goal. This is again done based on a probability weighted for patch attractiveness. The *surroundings* for the rangers are those patches within his *ranger-vision* that are inside the park and do not contain a camp. The attractiveness consists of rhino signs, poacher signs and roughness; using the user-entered weights for relative importance of looking for rhinos and/or poachers as well as avoiding roughness:

```
let surroundings patches in-radius ranger-vision with [not any? camps-here
      and inside? = TRUE]
 ask surroundings [
   let rhino-signs (recent-rhino-activity / (rhino-activity-decay * 2)) *
       search-rhino-weight
   let nearby-poacher-signs ((human-signs-decay - poacher-signs) / human-
       signs-decay) * catch-poacher-weight
   let estimated-roughness ((max-roughness - roughness) / max-roughness) *
       roughness-weight
   set patch-attractiveness (other-patrol-dist + rhino-signs + nearby-
       poacher-signs + estimated-roughness) / ranger-weight
   if patch-attractiveness < 0 [
      set patch-attractiveness 0 ]
]
```

Where *other-patrol-distance* uses the *ranger-avoid-weight* to keep patrols at some distance from one another:

```
to-report other-patrol-dist
    let nearby-rangers (turtle-set other rangers camps)
    ifelse any? nearby-rangers [
      let max-ranger-dist distance min-one-of nearby-rangers [distance
            myself]
      ifelse max-ranger-dist != 0 [
        report (distance min-one-of nearby-rangers [distance myself] / max-
            ranger-dist) * ranger-avoid-weight ]
    [ report 0 ]
    ]
  [ report 0 ]
end
```

Patches that have been recently visited (*patrol-memory*) are again made less attractive to avoid an infinite back and forth between two patches. Using `rnd:weighted-one-of` a goal is now selected based on *patch-attractiveness*. If that goal is directly adjacent to the current patch the ranger moves there right away. If the goal is further away in the *surroundings* the ranger moves towards it, also using `ranger-movement`.

3.3.2. Ranger-movement

Like rhinos and poachers the movement procedure uses the same rules as the checking and setting of the goal, only adding the distance and weight of the goal to the parameters that make up *patch-attractiveness*:

```
ask neighbors with [inside? = TRUE] [
    let goal-dist (max-goal-dist - distance [ranger-goal] of myself) * goal-
       weight
    let rhino-signs (recent-rhino-activity / (rhino-activity-decay * 2)) *
       search-rhino-weight
    let nearby-poacher-signs ((human-signs-decay - poacher-signs) / human-
       signs-decay) * catch-poacher-weight
    let estimated-roughness ((max-roughness - roughness) / max-roughness) *
       roughness-weight
    set patch-attractiveness (goal-dist +  estimated-roughness + other-
       patrol-dist + rhino-signs + nearby-poacher-signs) / (ranger-weight +
       goal-weight)
    if patch-attractiveness < 0 [
      set patch-attractiveness 0 ]]
  let surroundings neighbors with [not member? self [patrol-memory] of
      myself]
  ifelse any? surroundings [
    move-to rnd:weighted-one-of surroundings [patch-attractiveness] ]
  [ move-to rnd:weighted-one-of neighbors [patch-attractiveness] ]
```

The alternative patrol type is the *fence-patrol*, called when the ranger *state* is patrolling and the patrol type is set to fence patrol. Instead of calling `ranger-patrol`, `ranger-fence-patrol` is now called. During the fence patrol rangers move directly along the fence for the duration of their patrol. When poacher signs are found the `ranger-follow-up` procedure is called that makes rangers follow the tracks by moving to neighboring patches that have poachers signs equal to or greater than 1.

3.3.3. End of patrol and planning of new patrol

The patrol ends when time has run out (or earlier when a poacher has been arrested). The *state* is set to *back-to-camp* and the ranger moves to the nearest camp or border patch. The ranger has a waiting period similar to the laylow time of poachers, called *off-duty-time* (*(patrol-length * 2) + random 100*), which now starts counting down. Upon reaching 0 `ranger-new-patrol` is called.

The goal for this new patrol is selected from a patchset that includes the remembered *rhino-sightings* and *risky-areas*, and explicitly excludes *safe-areas*. This patchset is called *good-options*. If no such patches are available to a ranger a random patch that is not part of the *safe-areas* is selected and a start location is selected, like poachers planning a new trip. To select the starting point `ranger-start-location` is called, which selects the border or camp nearest to the set goal (unless it is a fence patrol, which always starts at the border). This procedure also resets patrol time, memory, off duty time and ranger goal as well as set the ranger's state back to *patrolling* to call the corresponding procedures allowing the next patrol to start.

The alternative is that *good-option* patches are available as a ranger has built his memory of rhino sightings and areas that are either risky or safe. Out of the *good-options* a patch is again selected as a goal, with probabilities weighted for *patch-attractiveness*. The *attractiveness* in this case is made as follows:

```
ask good-options [
       let detected-rhino-signs (recent-rhino-activity / (rhino-activity-
              decay * 2)) * search-rhino-weight
      let detected-poacher-signs ((human-signs-decay - poacher-signs) /
              human-signs-decay) * catch-poacher-weight

    set patch-attractiveness (detected-rhino-signs + detected-poacher-signs)
       / new-patrol-weight
    if patch-attractiveness < 0 [
      set patch-attractiveness 0 ]
```

Again a *best-option* is determined as the patch in the *good-options* with the highest *patch-attractiveness*. Allowing for some explorative behavior again this is compared to a random value

between 0 and 1. If the highest attractiveness is higher than this random value, the ranger selects as his goal one of the *good-options* within a 1.5 radius, using `rnd:weighted-one-of` to weight the attractiveness. The starting location is determined by calling `ranger-start-location` at this point. If the random value is greater than the *best-option* a random location is set as the target and the start location is again determined using the same procedure.

3.4. Patches

All behavior of the agents when running `go` has now been described. This behavior affects the environment, creating a dynamic model (i.e. by rhinos eating the resources that direct their movement, as a part of *patch-attractiveness* and the resources growing back after a while). Some of the patch variables change under "natural" influences or the influence of other agents as well, requiring commands for patches instead of agents. These procedures are `update-resources` and `update-activity-signs` (in addition to the previously described `update-home-range` and `update-visualization`). These procedures are also called by `go`.

Updating the resources manages the growth of resources, representing the natural occurrence thereof and also replenishing what rhino agents have grazed. All patches with resources have a countdown set as the *sprout-delay-time* plus a random value between 0 and 100. This counts down 1 for every passing tick. Upon reaching 0 the resources of that patch increase by 1% ($resources * 1.01$), and the countdown is reset. The regrowth is capped at 1, to keep the resource range within 0 to 1.
Finally updating the activity signs simply decreases the signs of rhino, poacher and ranger presence by 1 for each tick.

Literature

Lemieux, A. M. (2014). *Situational Prevention of Poaching*. London and New York: Routledge

Glossary

Border	Outermost patches that are colored black and given no resources to represent a hard border or fence for the animals.
Camp	Sedentary agents shown as red houses in the park. Functions as a base for rangers whom originate from the camps and return there on occasion. Camps can either be placed randomly, manually or in fixed locations (`build-camps` procedure). They exclude one another in a 10 patch radius, and make a radius of 4.5 patches around them devoid of resources.
Carcass-decay	Global countdown determining when a carcass disappears from a patch. A carcass (poacher sign) is left on a patch as a poacher kills a rhino. The time is set to 10000 ticks, matching the maximum duration of a simulation. As such a carcass will always remain visible.
Failure-sites	Patchset containing an individual poacher's visited patches with > 0 ranger signs. Accessible by that poacher and used in patch-attractiveness. Good- and failure sites overwrite one another on revisits.
Fence	See border.
Fence-patrol	Ranger movement procedure that makes them patrol along the park borders.
Forage-amount	Percentage with which resources on a rhino's patch diminish, to represent grazing (*((resources * ((100 - forage-amount) / 100))*). Set by the slider in the interface.
Goal-dist	Distance of individual agent to its goal patch. Used when the goal is not directly adjacent to its current patch, and incorporated in patch-attractiveness weighted probability of selection (`rhino/poacher/ranger-movement`). This also makes use of the user-entered weight of the goal, to allow setting relative importance of the variables in patch-attractiveness.
Good-options (poachers)	Patchset accessible by poachers, consisting of good- and success-sites of each individual poacher. Used to set patch-attractiveness to select a goal for a new trip. Best-option is the patch within this set with the highest patch-attractiveness.
Good-options (rangers)	Patchset accessible by rangers, consisting of rhino-sightings and risky-areas of each individual ranger. Used to set patch-attractiveness to select a goal for the next patrol. Best option is the patch within this set with the highest patch-attractiveness.
Good-sites	Patchset containing an individual poacher's visited patches with > 0 rhino and 0 ranger signs. Accessible by that poacher and used in patch-attractiveness. Good- and failure sites overwrite one another on revisits.
Human-signs-decay	Global countdown determining when poacher and ranger signs disappear from a patch. Set to 200 ticks in `setup-park`.
Laylow	Time for poachers to wait in between poaching trips. It is defined as *(poacher-time * 2) + random 100* and starts counting down when the poacher has left the park. Upon reaching 0 poacher-plan-new-trip is called to plan the following hunting trip.
Max-roughness	The maximum roughness value of 5 is given to the number of patches specified by the user using the rough-areas slider.
Nearby-rangers	Patchset accessible by poachers, consisting of all rangers and ranger camps within the individual poacher's vision-radius. Used for avoidance purposes.
Off-duty-time	Time for poachers to wait in between patrols. It is defined as *(patrol-length * 2) + random 100* and starts counting down when the ranger has come back from patrol. Upon reacher 0 ranger-new-patrol is called to plan a new patrol.
Other-patrol-distance	Distance between ranger agent and nearest other ranger agent. Weighted by the user-entered ranger-avoid-weight and used in patch-attractiveness. Allows rangers to avoid one another.
Patch-attractiveness	Variable created for a set of patches surrounding each agent (see *surroundings*). Consists of numerous variables for each agent type, to be found in the `rhino/poacher/ranger-decision-making` and `rhino/poacher/ranger-movement` procedures as well as `poacher-plan-new-trip` and `ranger-new-patrol`. Used to create weighted probability of selection of that patch as the next movement goal.

Patrol-memory	Memory of individual rangers, consisting of a patchset of recently visited patches; accessible by the individual ranger. Used to avoid moving back and forth between two patches.
Patrol-time	Individual time count for each ranger, keeping track of how long he has been on patrol. The maximum patrol time is set by the patrol-length slider, which when reached results in the ranger going back to camp.
Poacher-goal	Selected patch for the ranger to move to. The selection looks at the surroundings (see surroundings in this glossary) and uses a weighted probability based on patch-attractiveness to set the next goal. This is then moved to or towards by the poacher.
Poacher-memory	Patchset consisting those patches visited by the individual poacher during a hunting trip, used to avoid infinite revisits. The set is cleared upon a successful kill.
Poachers	Agents that enter the park periodically to look for rhinos.
Poacher-signs	As poachers move through a patch there is a 0.5 probability that a poacher sign is left there as a patch-owned variable (set in `setup-humans` as the human-signs-decay of 200 ticks, allowing a varying countdown). This is used in various movement procedures as part of patch-attractiveness. Sign decay is determined in `update-activity-signs`, which counts down the signs by 1 for every passing tick.
Poacher-vision	The radius of patches in which poachers can see rhinos and rangers (as well as their camps), used to kill and avoid them respectively. Also used as the poacher's *surroundings*.
Poacher-weight	Sum of user-entered weights for poacher movement procedure, set in `setup-park`: *(ranger-avoid-weight * 2) + hunt-rhino-weight + roughness-weight + resources-weight*. Used in `poacher-decision-making` and `poacher-movement` as part of patch-attractiveness.
Ranger-dist	Distance between poacher agent and nearest ranger agent. Weighted by the user-entered ranger-avoid-weight and used in patch-attractiveness. Allows poachers to avoid rangers.
Ranger-goal	Selected patch for the ranger to move to. The selection looks at the surroundings (see *surroundings* in this glossary) and uses a weighted probability based on patch-attractiveness to set the next goal. This is then moved to or towards.
Rangers	Agents that enter the park periodically to look for poachers.
Ranger-signs	As rangers move through a patch there is a 0.5 probability that a ranger sign is left there as a patch-owned variable (set in `setup-humans` as the human-signs-decay of 200 ticks, allowing a varying countdown). This is used in various movement procedures as part of patch-attractiveness. Sign decay is determined in `update-activity-signs`, which counts down the signs by 1 for every passing tick.
Ranger-vision	The radius of patches in which rangers can see poachers, and from which a goal is selected as these patches make up the *surroundings*.
Ranger-weight	Sum of user-entered weights for ranger movement procedure, set in `setup-park`: *1 + search-rhino-weight + roughness-weight + catch-poacher-weight*. Used in `ranger-decision-making` and `ranger-movement` as part of patch-attractiveness.
Recent-rhino-activity	As rhinos move through a patch there is a 0.5 probability that a rhino sign is left there as a patch-owned variable (set in `release-rhinos` as the rhino-activity-decay + random 500, allowing a varying countdown). This is used in various movement procedures as part of patch-attractiveness.
Resources	Patch variable ranging from 0 to 1 (1.1!). Foraged for by rhinos agents that diminish the amount of resources by a user determined percentage (`rhino-foraging-and-home-range`). Re-growing 1% every time the sprout-delay-time has run out (`rhino-foraging-and-home-range`). Resources are shown in shades of green where light is low and dark is high.
Rhino-activity-decay	Global countdown determining when rhino signs disappear from a patch. Base time is set to 500 ticks in `setup-park`, randomized by adding between 0 and 500 ticks in `release-rhinos`. Sign decay is determined in `update-activity-signs`, which counts down the signs by 1 for every passing tick.

Rhino-dist	Distance between rhino agent and nearest other rhino agent. Weighted by the user-entered rhino-avoid-weight and used in patch-attractiveness. Allows rhinos to avoid one another.
Rhino-goal	Selected patch for the rhino to move to. The selection looks at the surroundings (see *surroundings* in this glossary) and uses a weighted probability based on patch-attractiveness to set the next goal. This is then moved to or towards.
Rhino-memory	Memory of individual rhinos, consisting of a patchset of the last 3 visited patches accessible by the individual rhino. Used to exclude these as potential next movement goals, thereby avoiding long loops of back and forth moving between two patches.
Rhinos	Agents that live in the park and looks for resources.
Rhino-sightings	Patchset accessible by individual rangers consisting of those patches in which that ranger has seen a rhino (looking at his own and neighboring patches).
Rhino-territory	All patches within a 1.5 patch radius of the rhino agent are set as its territory, or home range. This makes those patches less attractive to other rhinos, but does not exclude them fully. As such home ranges can be overwritten.
Rhino-weight	Sum of user-entered weights for rhino movement procedure, set in `setup-park`: *rhino-avoid-weight + roughness-weight + resources-weight*. Used in `rhino-decision-making` and `rhino-movement` as part of patch-attractiveness.
Risky-areas	Patchset accessible by individual rangers containing an individual ranger's visited patches with > 0 poacher signs. Used in patch-attractiveness. Risky- and safe areas overwrite one another on revisits.
Roughness	Patch variable ranging from 1 to 5, representing terrain roughness and used to determine movement speed (in each agent's goal-check procedure). The user-specified number of rough areas is given the maximum value, which is then faded out, in `create-rough-areas`. Roughness is shown in shades of red where light is low and dark is high.
Safe-areas	Patchset accessible by individual rangers containing an individual ranger's visited patches with 0 poacher signs. Used in patch-attractiveness. Risky- and safe areas overwrite one another on revisits.
Sandy-areas	User determined percentage of patches that are made brown and barren, given no resources: *((count patches with [inside? = TRUE]) / 100) * sandy-areas*.
Sprout-delay-time	Global countdown determining when resources can regrow. The base time is set to 100 ticks in `setup-park`, which is randomized by adding a random amount of ticks between 0 and 100 in `create-park` and `update-resources`. This counts down 1 for every passing tick. Upon reaching 0 the resources on the patch increase by 1% (*resources * 1.01*).
Standard-patrol	Standard patrol deployed by rangers when this type has been selected, and described in the ranger-goal-check and ranger-movement.
State	Ranger and poacher owned variable, describing their current action. When needed the state is changed (i.e. from patrolling to off-duty-time), which then calls on the corresponding procedures to run (i.e. if state is off-duty-time …).
Success-sites	Patches where a poacher has been successful in killing a rhino, stored as a memory patchset of that individual poacher and used in patch-attractiveness, accessible by that poacher.
Surroundings	Patchset that is defined for each agent in its respective decision-making and movement procedures. It contains those patches from which the agent can select its next movement goal, based on relative patch-attractiveness within the set.
Tick	More or less abstract unit for time measurement, tracked in the tick-counter. It counts the number of plot updates (i.e an agent moves every X amount of ticks and patch variables are updated every Y amount of ticks). Simulations are set to run for a maximum of 10000 ticks.
Ticks-wait	Agent owned counter set to ticks-to-stay-on-patch. Makes agents wait for a number of ticks of roughness – 1, thereby simulating terrain that is harder to move through. It is in each agent's goal-check procedure, and described for rhinos on page 8.

Time (poacher)	Individual time count for each poacher, keeping track of how long he has been hunting. The maximum poaching time is set by the poacher-time slider, which when reached results in the poacher leaving the park via the shortest route to the nearest border.