PopSimCode some docs (unfinished)

Martin Emms

July 7, 2023

Contents

1 Overview

there are people (class `Person`, a collection of same is class `Population`)

there are food resources (class `CropPatch`, a collection same is class `Resources`)

there are locations (class `Location`)

there's an iteration over what could be thought of as days

as the days progress there are updates to the food resources, updates which are independent of anything people do. This is set up in such a way that food in a particular location has a kind of season, repeatedly rising to some max and falling to some min (actually 0).

as the days progress there are updates to the people and some of these updates affect the resources. There are at least the following kinds of updates:

ageing each day a Person gets older and there can be death due to old age; currently there is just a fixed max age

reproduction there is reproduction; currently after a certain reproductive age is reached, a Person might start to have off-spring and there's a particular probabilistic treatment of how many and when off-spring will be produced.

metabolism a Person is thought of as possessing a certain amount of energy and is thought of as necessarily **expending** energy every 'day' to stay alive (breathing in and out, pushing blood thru brain etc); currently this is set to be a fixed daily amount. Currently the code does not represent this energy expenditure as gradual: it is subtracted at the 'beginning' of a 'day' (which really just means before any updates relating to moving and feeding)

and if such a subtraction is not possible because the `Person` does not possess the energy to meet this daily enery expenditure they are deemed to die 'of starvation'.

feeding during a day all `Person`s seek to find food resources, and their consumption of food **adds** to their energy supply, so that sufficient food consumption will cause the avoidance of death by starvation. Currently this is implemented by an updating 'event queue', where the events concern things like movement and eating. Essentially this treats a single 'day' as an arbitrarily divisible time span, so that episodes of movement and eating can take arbitrary amounts of time. They are queued in the 'event queue' in the order of when they are due to occur. When they occur usually some state change takes place (a change of location, a change in the amount of energy possessed by a `Person`, a change in the amount of food remaining in a given `CropPatch` etc), and often one event spawns a further event to be added to the event queue. This has to be done is such a way that eventually the queue will become empty.

movement as things are currently implemented, each `Person` has a 'home' location, and during a 'day' they move away from their 'home' location, seek food, then return to their 'home' location.

The above seem a rough minimum 'biological' starting point for modelling populations and already allows instances of so-called '*exploitation competition*' to be studied. This is just the inevitable consequence that once one individual has consumed something, it can't also be consumed by another individual; the other individual will have to go somewhere else and/or wait till time passes till new resources appear. Note the way the modelling is set-up, the consumable food resources have a definite *maximum* possible amount, but there is no intrinsic ceiling to the number of `Person`s: depending how reproduction is configured, were the energy consumption requirements to be set to zero, the population would grow without limit. It is the food limit that leads to a population limit.

Besides this 'biological' minimum (and there are already plenty of choices and opportunites for refinement there), its possible then to also think about what happens when social/co-operative/communicative aspects are layered on top of this. As things stand the code currently has some initial implementations of some traits a `Person` may have relating to this:

- a `Person` may have the ability to notice and retain useful information about the food supply in particular locations, information which can effect how they 'forage' for food.
- a `Person` may have the ability to communicate such information

See Later

2 Design issues

```
[] currently there is a 'discrete time' loop -- an outer loop over
'days' -- and then within each 'day' iteration, a so-called 'discrete
event' loop, which has a quite different logic. Whether that is a
good idea is something to be pondered, and seemingly many systems do
everything with a 'discrete time' loop.  My personal hunch is
that the discrete time approach forces the model to adopt a time-step
size, probably setting this small to model successive states of a
fast process, but that it then needs to treat slow processes
laboriously as very long and expensive to implement chains of
micro-steps.
```

```
[] Persons and CropPatchs are treated as possessing a Location as an
attribute. The Locations themselves have 2-dimensional real-valued
coordinates, so there infinitely many potential locations, but the
model only creates a small *finite* 'network' of Locations: the set
of places where the small set of 'interesting' things are.  This is
done to make movement and navigation (arguably) simpler: there is not
an infinity of 'next' locations to move to.  Possibly other design
choices could be explored.
```

3 Details

3.1 Locations and geometry

Locations have coordinates and links to other Locations

The layout of locations is done by functions in *world_setup.cpp*.

Each Resources object has an associated collection of Locations (see Resources::locs) with the idea that at each of these Locations there are 1 or more CropPatchs. Currently the Locations of a Resources object are laid out on a horizontal line

```
(x,y) -> <- (x+d,y) -> <- (x+2d,y) -> ..... -> <-(x+(N-1)d,y)
```

for some number N of Locations, and distance d separating them : the particular x and y values in this sequence will vary from one Resources object to another. Every Location has a representation of other Locations that it is linked to (see Location::nbs), and the above Locations have links to their neighbours on the line (1 for terminal points on that line, 2 for internal points).

There could be several Resources objects: the code in *world_setup.cpp* currently sets up 5.

Locations are treated as being of one several different kinds (see class LocKind). Those belonging to a Resources object get kind PATCH

For each Resources area, besides the Locations in its loc member, it is associated with a 'resource entry' Location immediately to the left of the area's contained Locations:

```
(x-d,y) -> <- (x,y) ...
```

This Location has kind RES_ENTRY. It and the first Location of a Resources area are mutually linked.

So there are 5 such 'resource entry' Locations, because there are 5 Resources areas.

These 'resource entry' Locations are placed around a semi-circle, effectively at compass-points (N, NE, E, SE, S. At the centre of this circle is a 'hub' Location (kind is NODE) and to the west of this 'hub' there is a 'home' Location (kind is HAB_ZONE).

Fig 1 gives an impression of this geometry (Note the locations within the Resources areas associated with an entry are only indicated for the first area : in each case they are to the right of the entries).

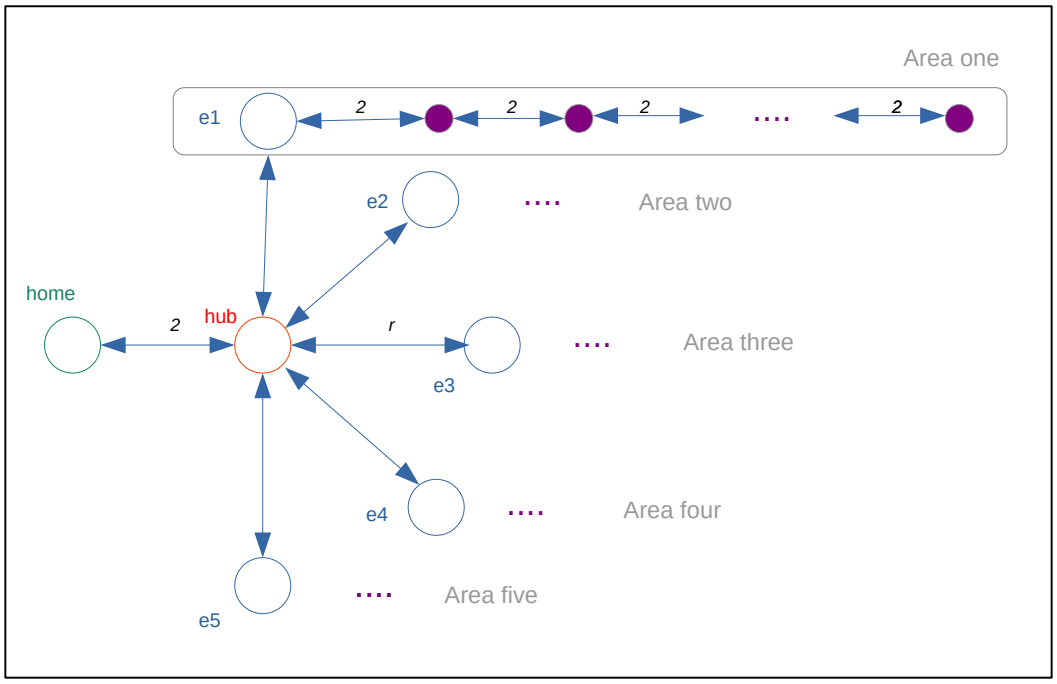


Figure 1: 5 `Resource` areas, the locations and linking of their entries, the 'hub' and 'home'

As far as linking goes:

- the central 'hub' and the 'resource entry' locations are mutually linked
- the 'home' and the 'hub' locations are mutually linked
- the 'resource entry' locations are *not* mutually linked

As far as distances go, 'home'-'hub' distance is 2, as is the distance between all adjacent locations belonging to an area. The 'hub'-'entry' distance r is 2.613126[1]

If the coords of the 'hub' loc were at $(0, 0)$, then for $a = 1/\sqrt{2}$, the coords of the entries *would* be $(0, r)(a, a)(r, 0)(a, -a)(0, -r)$. For not very good reasons to do with a possibility of Qt visualisation, the coordinates actually used transform these coordinates according to $(x, y) \mapsto (x, 3 - y)$: this means the 0 value of y is to be pictured at the top in Fig 1, with increasing y going down the page. The coords of the 'hub' loc are at $(0, 3)$, and the rest work out as:

```
                        e0  0,3-r  [ = (0.0, 0.3868741) ]

                            e1 a,3-a  [ = (1.847759, 1.152241) ]

(-2,3) home    (0,3) n              e2 r,3  [ = (2.613126, 3) ]

                            e3 a,3+a [ = ( 1.847759, 4.847759) ]

                       e4  0,3+r [ = (0, 5.613126) ]
```

On the current set-up, the network of locations will be reported as follows

[1]This awkward seeming number is the value which will entail that the 'resource entries' are a (straight-line dist) of 2 apart: this used to be a convenient choice when there were inter-entry links, which there no longer are.

```
h[-2,3] - node - n[0,3]
n[0,3] - home - h[-2,3]
n[0,3] - res_entry - e[0,0.386874]
n[0,3] - res_entry - e[1.84776,1.15224]
n[0,3] - res_entry - e[2.61313,3]
n[0,3] - res_entry - e[1.84776,4.84776]
n[0,3] - res_entry - e[0,5.61313]
e[0,0.386874] - node - n[0,3]
e[0,0.386874] - res - c[2,0.386874]
e[1.84776,1.15224] - node - n[0,3]
e[1.84776,1.15224] - res - c[3.84776,1.15224]
e[2.61313,3] - node - n[0,3]
e[2.61313,3] - res - c[4.61313,3]
e[1.84776,4.84776] - node - n[0,3]
e[1.84776,4.84776] - res - c[3.84776,4.84776]
e[0,5.61313] - node - n[0,3]
e[0,5.61313] - res - c[2,5.61313]
c[2,0.386874] - res_entry - e[0,0.386874]
c[3.84776,1.15224] - res_entry - e[1.84776,1.15224]
c[4.61313,3] - res_entry - e[2.61313,3]
c[3.84776,4.84776] - res_entry - e[1.84776,4.84776]
c[2,5.61313] - res_entry - e[0,5.61313]
```

where a `Location` objects is reported with h/n/e/c according to its kind ('home', 'node', 'entry', 'crop patch)

Note: one of the outputs of a simulation run is *sim_setup*, a subsection of which records the above kind of 'geometry' information about the `Location`s

Important Global Vars

`vector<LocPtr> all_res_entry_loc` : (pointers to) the `Location`s which are 'resource area entry': currently there are 5 of these

`vector<LocPtr> all_home_loc` : (pointers to) the `Location`s which are 'home' locations: currently there is just one of these

3.2 Resources: classes `CropPatch`, `Resources`

3.2.1 `CropPatch`

Food is available in a `CropPatch`.

Could be thought of as a food source that a `Person` can monopolise while they are eating from it, eg. a bush with berries on it.

As time passes it provide a varying amount of food 'units' in a periodic way. The total amount it will in time provide is determined by `CropPatch::patch_yield`. `CropPatch::profile` represents how much of this patch's yield becomes *first* edible ('ripens') at given point through its period or 'season'. For example a `profile` of $\{0.05, 0.1, 0.2, 0.3, 0.2, 0.1, 0.05\}$ indicates that there are 7 successive days on which particular parts of the patch (units within it) *first* become edible, and the number indicate the percentages that 'ripen' on each day.

Food remains edible for a while and `CropPatch::lasts` gives the number of days this is the case (this includes the day it first becomes edible). For example, combining above `profile` with `lasts==4` gives a patch which will have food for 10 successive days: 7 'new' growth days with 3 days after that before the most recent food becomes inedible.

The whole process for a given `CropPatch` repeats, and `CropPatch::period` is the length of the cycle: can be thought of as number of days between repeat instances of `profile[0]` amount of the crop becoming first edible. Note this may well leave a number of food-free days after the most recently ripened parts have become too old.

Finally there is `CropPatch::abs_init_day` to fix when this periodic behaviour starts

As an example, suppose a `CropPatch` has characteristics

```
profile = {0.05, 0.1, 0.2, 0.3, 0.2, 0.1, 0.05}
lasts = 4
patch_yield = 100
abs_init_day = 4
period:13
```

Below is a display of how the units of food available from it will wax and wane. The display shows units of food in 'age' bands – as $Age(Amount)$ – with oldest at left, youngest at right. At the end of the line is the total over 'age' bands

Day	Bands	Total	Notes
1	none		
2	none		
3	none		
4	1(5)	5	first 'new' growth ie. becomes edible that day
5	2(5) 1(10)	15	
6	3(5) 2(10) 1(20)	35	
7	4(5) 3(10) 2(20) 1(30)	65	first day with age 4; disappears on following day
8	4(10) 3(20) 2(30) 1(20)	80	
9	4(20) 3(30) 2(20) 1(10)	80	
10	4(30) 3(20) 2(10) 1(5)	65	last day with 'new' growth
11	4(20) 3(10) 2(5)	35	
12	4(10) 3(5)	15	
13	4(5)	5	final age 4 going to disappear
14	none		a first of several empty days
15	none		
16	none		
17	1(5)	5	starts over with 'new' growth again: its a cycle of length 13
18	2(5) 1(10)	15	
:	:		

So there is 'cycle' or 'period' or 'season' of length 13, with 10 food-containing days, followed by 3 food-free days. Over the first 7 days varying amounts of become edible for the first time, and having done so remain in each case for a total of 4 days. The result as indicated by the totals is a rising then falling pattern over 10 days, which then flat-lines at 0 for 3 days, before starting up all over again.

Each `CropPatch` has a `Location`. Potentially each could have its own unique characteristics.

In the current implementation, all `CropPatch`s are of two types

One type ('berries') has

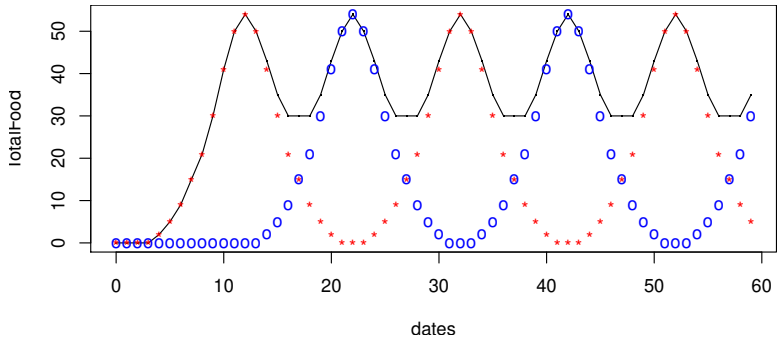
```
abs_init_day:4
profile:0.02,0.03,0.04,0.06,0.08,0.12,0.15,0.15,0.12,0.08,0.06,0.04,0.03,0.02
patch_yield:100
lasts:4
period:20
```

The other type ('beans') has essentially the same characteristics except for

```
abs_init_day:14
```

This has the effect that repeatedly as the 'berries' season is starting to peter out, the 'beans' season is starting to pick up.

The plot below shows crop totals that would result from having 1 of each type of `CropPatch` (black solid line is total of both, red (*) is 'berries', blue (o) is 'beans')



Each is given a display symbol to assist with displaying crop amounts `*` for 'berries', `o` for 'beans'.

TODO: some info on `CropPatch::bands`, `CropPatch::update_at_date(..)`

The main (only?) significance of a `CropPatch` is that it may provide food. As mentioned in 1 a `Person` is thought of as obtaining *energy* from food.

There is a conversion rate `CropPatch::energy_conv` from crop 'units' to 'energy' units, so one potentially can have `CropPatch`s with higher and lower yields of energy for the same amount of crop 'units' (currently all have `energy_conv` set to 0.25).

Most of the functions relating to eating are parts of `Person` and `Population`. The eating processes, however, are all organised around an 'eat oldest units first' principle. The function `CropPatch::remove_units(int units)` does an update to a `CropPatch` which diminishes the units available in its bands, taking away the units of the oldest bands first. See further under `Person` and `Population`; its only called when its already been determined that the `CropPatch` contains at least the indicated number of units.

3.2.2 `Resources`

A `Resources` object is a kind of collection of `CropPatch`s and could be thought of as an 'area' in which particular `CropPatch`s are to be found. As mentioned in 3.1 each `Resources` 'area' has an associated collection of `Location`s (see `Resources::locs`) with the idea that at each of these `Locations` one can think of there being one or more `CropPatch`s.

Currently this is implemented by having `Resources::resources` be a vector of `CropPatch`s; the `Location`s of the `CropPatch`s coincide with `Resources::loc`. If the `Resources` area had 10 `Location`s and at each there were supposed to be 4 `CropPatch`s, the `Resources::resources` vector would have 40 entries and each `Location` will be mentioned by 4 different `CropPatch`s in this vector.

Currently a `Resources` area has to have a set of `Locations` which stand in a particular relation to an associated 'resource entry' `Location` (see 3.1), and this is currently taken care of by a `Resources` area constructor which is passed the 'resource entry' `Location`. The detailed configuration of the area (the number of `Location`s, the number of `CropPatch`s at those locations, the precise nature of those `CropPatch`s) is currently written into the `Resources` area constructor.

Note: one of the outputs of a simulation run is *sim_setup*, and a subsection of this records the configuration of `Resources` areas.

Important Global Vars

`vector<ResPtr> all_res` contains pointers to all the `Resources` objects. Currently these are dynamically allocated by code in *world_setup.cpp*

`map<LocPtr, ResPtr> loc_to_res` a `Resources` area contains its `Location`s via `Resources::locs` but it necessary sometimes go from a `Location` to the `Resources` area it is a part of and the map `loc_to_res` allows that. Movement is from `Location` to `Location` and this is used in `Population::update_by_move_and_feed(..)` to determine the relevant `Resources` area from a `Location`

3.3 People: classes `Person`, `Population`

class `Population` is intended to represent a collection of `Person`s. The constituent `Person`s undergo changes every 'day'. These changes are implemented through a series of `Population::update_****(...)` methods. So certain 'scheduling' aspects of the simulation are settled by how these methods are called, which in outline is:

```
Population::update(..)

 --> Population::update_by_cull(..) // age, expend energy, cull

 --> Population::update_by_move_and_feed(..)

 --> Population::update_by_repro(..)
```

`Population::update(..)` is called once on each 'day' of a simulation run. `update_by_cull` and `update_by_repro` are executed by single pass through of all the contained `Person`. This does not imply any finer temporal subdivision than the 'day' level and thus all follows the 'discrete time step' approach to simulation, as discussed p111 [1].

`Population::update_by_move_and_feed(..)` is different, and is not executed by a single pass through all the contained `Person`s. It *initialises* an 'event queue' by such a single pass, with per-`Person` events, each containing a time which can be specified to any arbitrary precision. The events are sorted by these times, are taken from the queue in that order and treated as if these are times at which particular some process comes to an *end*, with one event typically spawning descendant events. Thus this part treats a single 'day' as an arbitrarily divisible time span and follows the 'continuous time' approach to simulation, as discussed p111 [1].

`Person` has members relating to age:

```
int Person::age
int Person::expiry_age
```

and part of `Population::update_by_cull(..)`'s execution to increment everyone's `age` by 1, and remove anyone whose age exceeds their `expiry_age` ie. they die of old age (currently everyone's `expiry_age` is set to 500)

`Person` has several members relating to 'energy'

`float Person::current_energy` is a `Person`'s 'energy', which can vary up and down as time passes. As time passes energy will inevitably be expended, and as time passes this may be compensated by consumption of food, which converts to energy.

The expenditure side is determined by `float Person::daily_use`, to be thought of as an inevitable amount of energy expenditure in a single 'day'. This is held constant, so its a kind of an idealisation, treating someone's metabolism and other energy demands (eg. movement) as always amounting to the same every day.

The other part of what `Population::update_by_cull(..)` does is to check whether a `Person`s `current_energy` is $\geq$ than the set value of their `daily_use`. If its the case, their `current_energy` is reduced by their `daily_use`, and if its not case, they are removed from the `Population` and adjudged to have starved. Note there is a somewhat arbitrary 'accountancy' aspect to this, with a `Person`'s daily energy expenditure all 'accounted for' at the 'beginning' of the day. Without a more complex implementation of gradual expenditure, such arbitrariness will remain.

`Population::update_by_move_and_feed(..)` is the implementation of how all members of the `Population` move about and consume food in a single 'day', which depends on the geometry of `Resources` areas, the available food in those areas and on some further energy related attributes of a `Person`

```
float Person::max_energy; // how much can energy can be stored up at most
float Person::max_daily_eat; // cannot cram more food in (as energy) in a single day
```

The idea is that a `Person`'s body stores energy that has been derived by eating food (as fat, blood-sugar etc) and `Person::max_energy` is the maximum level that can be reached. The implementation calls off further attempts to find food if this level is reached (currently this max is 7).

Also `Person::max_daily_eat` is the maximum increment of energy a `Person` could possibly manage to add to their energy via food consumption in a single 'day' (currently this is 3.5). This has the effect that if someone's `current_energy` has fallen to a very low level it could take several days, even in the most benign circumstance, for them to reach `max_energy`.

There are a lot of detailed aspects of food finding and consumption in `Population::update_by_move_and_feed(..)`, but beyond the details, there's an overarching design aspect:

a `Person` *continues to strive to find and consume food until*
either (a) they reach one of the ceilings `max_energy` *or* `max_daily_eat`
or (b) there is no more food to be found

In alternative implementations it would be possible to deviate from this 'strategy'

In 3.4 some of the further 'event queue' details involved in `Population::update_by_move_and_feed(..)` are outlined.

3.4 Events: classes `Event`, `EventLoop`

There are several kinds of 'event', defined by inheritance from a class `Event`. The derived classes set a member `kind` to indicate which kind of event it is, which ultimately allows the event queue to contain `Event` pointers, storing addresses of objects of classes derived from `Event`. The kinds of event are:

`ARRIVE`, `END_EAT`, `END_STAGE`, `END_REST`, `END_WAIT`

and the two most important kinds for the eating process are `ARRIVE` and `END_EAT`

- about an `ARRIVE` event

  1. The processing of an `ARRIVE` event `(t, p, l)` usually leads to the posting of `END_EAT` event `(t+h, p, [+g,-u])` where
     `g` is the energy `p` will gain by eating `u` units from a particular patch at `l`
     `h` is the handling time `hrate * u` for that number of units
  2. When there are several 'available' patches at l, one is randomly chosen
  3. An `ARRIVE` event (for person p) will not lead to such an `END_EAT` event if there is no 'available' food at l at that time. This happens not just because all patches at the location have no units left, but can well happen because some of the patches are *being eaten*, by other people who arrived *earlier* at the location and started to eat from the patches at the location. The fact of the patch being eaten is recorded in the `CropPatch::being_eaten` attribute, whose setting is a further consequence of an `END_EAT` event having been posted (for another person p').
  4. When an `ARRIVE` event does not lead to a `END_EAT` event, it will usually lead to the posting of an `ARRIVE` event to take someone to a next `Location` where there are `CropPatch`s

- about an `END_EAT` event

  1. At the time of *posting* the `END_EAT` event, the function `Person::eat_from_dry_run` is used to determine the anticipated energy gain for the eater, and associated number of units which will go from the patch. This is done in such a way that the Person strives to eat as much as possible, given their `max_energy` and `max_daily_eat`. This also determines the 'handling' time it will take to consume the anticipated number of units.

2. The increment `+g` to the eater's energy, and the decrement `-u` to the units of the patch eaten from do not happen at the time the `END_EAT` event is posted, but happen when that event in its turn is *processed*. This might be soon if only a small number of units is due to be eaten, but could be rather later if a large number of units is due to be eaten. The decrement to the units of the patch is done by the function `CropPatch::remove_units(int units)` (this was mentioned in 3.2.1) which diminishes the units available in its bands, taking away the units of the oldest bands first.
3. After these updates there are some different possibilities for an event which will be consequently posted.

   If they have not yet reached the limit imposed by `max_energy` or `max_daily_eat`, they will try to eat again. If this is possible at the current location, this will lead to a further `END_EAT` event, and if it is not possible, this will probably lead to an `ARRIVE` to take someone to a next `Location` where there are `CropPatch`s (there are other possibilities).

Fig 2 gives further details of the processing of the different kinds of events. Each event e_1 at the left represents an event about to be *processed*; those to the right reached by following arrows are events which are *posted* as part of the processing of the event e_1. Where the path from e_1 has a blue node in the middle, to the left of this node the *state updates* associated with processing e_1 are briefly noted. The text to the right of the posted events says what will be accomplished when they are in their turn processed. Note that when the path splits into several continuations, there are conditions which pick just one continuation.

3.4.1 A detailed miniature example of use of the event loop

Below the functioning of the event queue is illustrated

In this example, there is just 1 `Resources` area, which contains 3 locations. At the locations are 2 patches, one '*' and one 'o', with the kind of characteristics outlined in 3.2.1. The `patch_yield` is 50, and the example assumes that the first day on which eating occurs is day 60. The population is set to contain just 3 `Person`s.

If the code is run with certain debug settings activated, the situation before eating begins will be display somewhat like this:

```
60 STARTS
res:
one(2,y)
*c[2,y]:4(1) 1
*c[4,y]:4(1) 1
*c[6,y]:4(1) 1
oc[2,y]:4(3) 3(4) 2(6) 1(7) 20
oc[4,y]:4(3) 3(4) 2(6) 1(7) 20
oc[6,y]:4(3) 3(4) 2(6) 1(7) 20

people:
pop size: 3

person: 2A age: 334 rep: 0 Eout: 2 mxEn: 7 mxEat: 3.5 En:3 Eaten: 0
person: 1A age: 167 rep: 0 Eout: 2 mxEn: 7 mxEat: 3.5 En:3 Eaten: 0
person: 0A age: 1 rep: 0 Eout: 2 mxEn: 7 mxEat: 3.5 En:3 Eaten: 0
```

Too save space, outputs below just writing y for the N-S coord of patch locations, as they all have same value (0.386874)

After the call to `Population::update_by_cull(..)`, the event queue processing all occurs via the call to `Population::update_by_move_and_feed(..)`

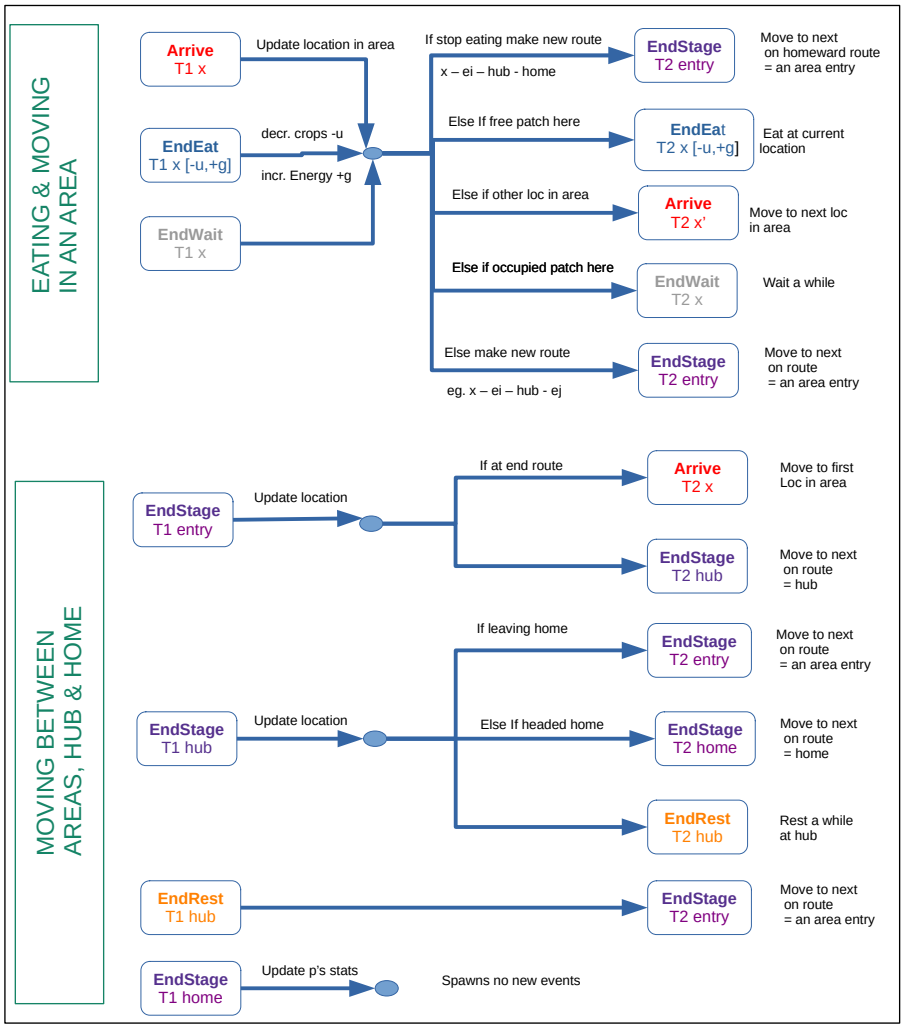


Figure 2: How *processing* different kinds of events leads to *posting* further events

Initially all are at 'home' location (the number after : shows number of people at the particular location)

```
h[-2,3]: 3 0A 1A 2A
```

and the event queue will gets initialised with `END_STAGE` events referring to the 'hub' location.

```
(1 per:1A end-stage n[0,3]) (1 per:2A end-stage n[0,3]) (1 per:0A end-stage n[0,3])
```

These events share the same time due to the individuals having the same speed and destination: there are some elements of randomisation in the code putting simulataneous events into an order.

The processing of each of these relocates the individuals to the 'hub' location and posts further `END_STAGE` events referring to the single entry location:

```
h[-2,3]: 0
n[0,3]: 3 0A 1A 2A

(2.30656 per:1A end-stage e[0,y]) (2.30656 per:2A end-stage e[0,y]) (2.30656 per:0A end-stage e[0,y]
```

The processing of each of these relocate the individuals to the entry location and posts further ARRIVE events referring to first crop patches to be reached[2]:

```
h[-2,3]: 0
n[0,3]: 0
e[0,y]: 3

(3.30656 per:1A reach c[2,y]) (3.30656 per:2A reach c[2,y]) (3.30656 per:0A reach c[2,y])
```

The first event is taken from the queue and processed, causing various state changes and the posting of further events. Specifically:

1A is now officially at `c[2,y]`
there are 2 (non-empty & unoccupied) patches there, which will be noted as:
`#patches` 2 at `c[2,y]`
randomly chooses `*c[2,y]`, which has 1 unit of food
this posts an `END_EAT` event for 1A : `(3.41656 per:1A ate 0.25 [1])`
its completion time is `3.30656 + (0.11 * 1) = 3.41656` because `hrate=0.11`
so now have

```
e[0,y]: 2
c[2,y]: 1 eating:1A
c[4,y]: 0
c[6,y]: 0

(3.30656 per:2A reach c[2,y]) (3.30656 per:0A reach c[2,y]) (3.41656 per:1A ate 0.25 [1])
```

processing the first event,
2A now officially at `c[2,y]`
now `#patches` 1 at `c[2,y]`
so has to choose `o[2,y]`, which has 20 units of food
as 14 units gives 3.5 which is `max_daily_eat` posts an `END_EAT` for 2A: `(4.84656 per:2A ate 3.5 [14])`
its completion time is `3.30656 + (0.11 * 12) = 4.84656`
so now have

```
e[0,y]: 1
c[2,y]: 2 eating:1A 2A
c[4,y]: 0
c[6,y]: 0

(3.30656 per:0A reach c[2,y]) (3.41656 per:1A ate 0.25 [1]) (4.84656 per:2A ate 3.5 [14])
```

processing the first event,
0A now officially at `c[2,y]`
`#patches` 0 at `c[2,y]`
so 0A plan: `c[2,y] --> c[4,y]`
so posts an `ARRIVE` event for 0A: `(4.30656 per:2A reach c[4,y])`
note the timing of the event relates to distance to travel (2) and a `Person`'s `speed` (here 2 units of distance, per 1 unit of time)
so now have

```
e[0,y]: 0
c[2,y]: 3 (0A) eating:1A 2A
c[4,y]: 0
```

[2]The event processing thus far was confined to the lower part of Fig 2. The processing of these `ARRIVE`-type events will involve the upper part of Fig 2

```
c[6,y]: 0

(3.41656 per:1A ate 0.25 [1]) (4.30656 per:0A reach c[4,y]) (4.84656 per:2A ate 3.5 [14])
```

Have shown (0A) to remind that conceptually though 0A's location is `c[2,y]` it is really in transit to `c[4,y]`. Processing the first event,
1A's energy increments by 0.25; `*c[2,y]` decremented by 1
as other patch at `c[2,y]` being eaten (by 2A) there are `#patches` 0 at `c[2,y]`
so 1A plan: `c[2,y] --> c[4,y]`
so post an `ARRIVE` event for 1A: `(4.41656 per:1A reach c[4,y])`
note will be slightly later than 0A cos spent some time eating
so now have

```
c[2,y]: 3 (0A,1A) eating:2A
c[4,y]: 0
c[6,y]: 0

(4.30656 per:0A reach c[4,y]) (4.41656 per:1A reach c[4,y]) (4.84656 per:2A ate 3.5 [14])
```

processing first event,
0A now at `c[4,y]`
`#patches` 2 at `c[4,y]`
randomly chooses `*c[4,y]` which has 1 units of food
this posts an `END_EAT` for 0A: `(4.41656 per:0A ate 0.25 [1])`
so have

```
c[2,y]: 2 (1A) eating:2A
c[4,y]: 1 eating:0A
c[6,y]: 0


(4.41656 per:1A reach c[4,y]) (4.41656 per:0A ate 0.25 [1]) (4.84656 per:2A ate 3.5 [14])
```

1A now at `c[4,y]`
`#patches` 1 at `c[4,y]`
so has to choose `oc[4,y]`, which has 20 units of food
so post an `END_EAT` for 1A: `(5.84656 per:1A ate 3.25 [13])`
having already eaten 1 unit (0.25) of food, eating 13 more will take its consumption to its `max_daily_eat` of 3.5,
so now have

```
c[2,y]: 1 eating:2A
c[4,y]: 2 eating:0A 1A
c[6,y]: 0

(4.41656 per:0A ate 0.25 [1]) (4.84656 per:2A ate 3.5 [14]) (5.84656 per:1A ate 3.25 [13])
```

processing first event,
0A's energy increments by 0.25, `*c[4,y]` decremented by 1
as other patch at `c[4,y]` being eaten (by 1A) there are `#patches` 0 at `c[4,y]`
so 0A plan:`c[4,y] --> c[6,y]`
so post an `ARRIVE` event for 0A: `(5.41656 per:0A reach c[6,y])`
so have

```
c[2,y]: 1 eating:2A
c[4,y]: 2 (0A) eating:1A
c[6,y]: 0

(4.84656 per:2A ate 3.5 [14]) (5.41656 per:0A reach c[6,y]) (5.84656 per:1A ate 3.25 [13])
```

processing first event,
2A's energy increments by 3.5; `oc[2,y]` decremented by 14
as 3.5 is `max_daily_eat`, 2A is finished
an event is posted relating to travelling home – not giving extensive commentary on these
so have

```
c[2,y]: 1 (2A)
c[4,y]: 2 (0A) eating:1A
c[6,y]: 0

(5.41656 per:0A reach c[6,y]) (5.84656 per:1A ate 3.25 [13]) (5.84656 per:2A end-stage e[0,y])
```

processing first event,
0A now at `c[6,y]`
`#patches` 2 at `c[6,y]`
randomly chooses `*c[6,y]` which has 1 unit
posts `(6.84656 per:1A ate 3 [12])`
so have

```
c[2,y]: 1 (2A)
c[4,y]: 1 eating:1A
c[6,y]: 1 eating:0A

(5.52656 per:0A ate 0.25 [1]) (5.84656 per:1A ate 3.25 [13]) (5.84656 per:2A end-stage e[0,y])
```

processing first event,
0A's energy incrments by 0.25; `*c[6,y]` decremented by 1.
there is `#patches` 1 at `c[6,y]` so chooses `oc[6,y]` which has 20 units
so posts an `END_EAT` for 0A: `(6.84656 per:0A ate 3 [12])`
note going to eat 12 units, which will take energy consumed to `max_daily_eat`
so have

```
c[2,y]: 1 (2A)
c[4,y]: 1 eating:1A
c[6,y]: 1 eating:0A

(5.84656 per:1A ate 3.25 [13]) (5.84656 per:2A end-stage e[0,y]) (6.84656 per:0A ate 3 [12])
```

processing first event,
1A's energy increments by 3.25, `oc[4,y]` decrements by 13
has reach `max_daily_eat` so post events relating to travelling home
so have

```
c[2,y]: 1 (2A)
c[4,y]: 1 (1A)
c[6,y]: 1 eating:0A

(5.84656 per:2A end-stage e[0,y]) (6.84656 per:0A ate 3 [12]) (7.84656 per:1A end-stage e[0,y])
```

processes the home-ward event, which lead to 2A not being at `c[2,y]`

so have

```
e[0,y]: 1
c[2,y]: 0
c[4,y]: 1 (1A)
c[6,y]: 1 eating:0A

(6.84656 per:0A ate 3 [12]) (7.15313 per:2A end-stage n[0,3]) (7.84656 per:1A end-stage e[0,y])
```

0A finishes eating from `oc[6,y]`
and posts home-ward travelling event
so have

```
e[0,y]: 1
c[2,y]: 0
c[4,y]: 1 (1A)
c[6,y]: 1 (0A)

(7.15313 per:2A end-stage n[0,3]) (7.84656 per:1A end-stage e[0,y]) (9.84656 per:0A end-stage e[0,y]
```

and after there is only processing of events relating to homeward travel:

```
(7.84656 per:1A end-stage e[0,y]) (8.15313 per:2A end-stage h[-2,3]) (9.84656 per:0A end-stage e[0,y

(8.15313 per:2A end-stage h[-2,3]) (9.15313 per:1A end-stage n[0,3]) (9.84656 per:0A end-stage e[0,y

(9.15313 per:1A end-stage n[0,3]) (9.84656 per:0A end-stage e[0,y])

(9.84656 per:0A end-stage e[0,y]) (10.1531 per:1A end-stage h[-2,3])

(9.84656 per:0A end-stage e[0,y]) (10.1531 per:1A end-stage h[-2,3])

(10.1531 per:1A end-stage h[-2,3]) (11.1531 per:0A end-stage n[0,3])

(11.1531 per:0A end-stage n[0,3])
```

leading to the situation after the last event is processed

```
h[-2,3]: 3 0A 1A 2A
n[0,3]: 0
e[0,y]: 0
c[2,y]: 0
c[4,y]: 0
c[6,y]: 0
```

concerning the population, the net effect of this round of eating is:

```
pop size: 3

person: 2A age: 334 rep: 0 Eout: 2 mxEn: 7 mxEat: 3.5 En:6.5 Eaten: 3.5
person: 1A age: 167 rep: 0 Eout: 2 mxEn: 7 mxEat: 3.5 En:6.5 Eaten: 3.5
person: 0A age: 1 rep: 0 Eout: 2 mxEn: 7 mxEat: 3.5 En:6.5 Eaten: 3.5
```

where all have eaten 3.5 and reached energy 6.5.

concerning the `CropPatch`s in the single `Resources` area, the situation now is

```
*c[2,0.386874]:none
*c[4,0.386874]:none
*c[6,0.386874]:4(1) 1
oc[2,0.386874]:1(6) 6
oc[4,0.386874]:1(6) 6
oc[6,0.386874]:2(1) 1(7) 8
```

If compared to the situation at the beginning of the day, 42 units have been taken away from the various patches (this is 3×14, which corresponds to the number of units for each to have gained 3.5 in energy. The oldest-first eating pattern is also visible.

afadsf

References

[1] Volker Grimm and Steven F. Railsback. *Individual-based Modeling and Ecology:*. Princeton University Press, 2005.