WML Filtering

Filters are a very important part of WML language, and a fairly complex too for various reasons. Here, we shall try to explain how to use them with more details than in the reference wiki pages. But the goal is not to replace these pages, and we assume you have at least some knowledge of the various WML filters, namely unit and location filters. The examples given below aren't always the best way to do things: the goal here is understanding filtering, not to give a complete WML course.

Basics: How filters work.

Filtering is narrowing a set of objects to a result set using criteria. Let's give an example : given a set of cards, if one is asked to select the spades, (s)he will probably check the cards one by one and create two stacks : one containing only the spades, and another containing the unselected cards. This is a very simple filter, where the criterion is « this card has spades color » and the result set is a card stack. The spades cards are said 'matching' the filter.

If next we're asked to find the king of spades, most probably, we will not restart from the beginning but only scan the spades stack to find the right card. This is an example of two criteria filter. In WML we would write something like :

```
[filter]
      color=spades
      value=king
[/filter]
```

to describe the operation. Criteria are expressions evaluating to true or false. Is this card color spades ? That's how we must read the sentence: 'color=spades'.
Now, stating both criteria must be met is not the only way to combine them:

```
[filter]
      color=spades,diamonds
      value=king
[/filter]
```

the first expression will be true if a card color is 'spades OR diamonds'. It would make no sense to state they should be 'spades AND diamonds', of course.
In many languages, you must specify how criteria combine using the special keywords 'OR' and 'AND'. In WML, you can do so, but most often, you're not requested to do so. Writing explicitly the logical operators would give something like that:

```
[filter]
      [and]
            color=spades
            [or]
                  color=diamonds
            [/or]
      [/and]
      [and]
            value=king
      [/and]
 [/filter]
```

or in natural speech: “is card (color=spades OR color=diamonds) AND value=king ?” Note here the use of parenthesis. As in algebra, they mean their content must be evaluated prior to apply the last criterion.
[and] and **[or]** subtags are WML equivalents of parenthesis. They are not mandatory (like in some other languages), and this is a cool feature, but can be misleading in complex filters. The rule is:

- listed criteria are ANDed, in other words, they must all be true for the object to match the filter.
- comma separated lists in a criterion are equivalent to ORed criteria. In other words, one only is enough for the object to match the filter.

There are some things important to note :

- A complex filter can always be split into simpler filters applied in chain, each filter taking as starting set the result set of the former one. In our example, we applied the filter « value=king » to the result set of filter « color=spades ». This is important when building or debugging filters, because complex ones can easily be reduced to a chain of simpler ones.

- Criteria order is not important from a logical point of view. We could have searched the kings first, obtaining the four kings in our result set, and the card of color spade next. The final result is the same. But in WML, for some reasons we shall study later, order can be important.

- Result sets can contain any number of objects. One can’t assume the result set of our filter will contain a single card ‘THE king of spades’. A human being would probably stop the search when finding a king of spades, but filters don’t. If our starting card packet is not a complete set (some cards were lost, a cheater introduced some more, or anything else), you’ll find one, none or many kings of spades. It’s a common error in WML to assume filters will select a single object[1].

- Any unit or location will match an empty filter like this one:

```
[event]
      name=moveto
      [filter]
      [/filter]
...
[/event]
```

This event will be triggered on every move of every unit on the map.

[1] One can be sure only when filtering with ID’s because they are unique.

Here now are two versions of the same action, using filters and not.

```
[modify_unit]
      [filter]
            side=2
            [not]
                  race=orc
            [/not]
      [/filter]
      side=1
[/modify_unit]
```

This moves to side 1 all side 2 units except orcs. Please note how filter syntax is after all very close to natural speech. This is why we shall see a good way to design complex filters is writing an accurate sentence describing them first.
Using no filter (and assuming 'all_units' is an array containing all created units in a scenario) we should write :

```
{FOREACH all_units i}
      [if]
            [variable]
                  name= all_units[$i].side
                  equals=2
            [/variable]
            [and]
                  [variable]
                        name= all_units[$i].race
                        not_equals=orc
                  [/variable]
            [/and]
            [then]
                  [set_variable]
                        name= all_units[$i].side
                        value=1
                  [/set_variable]
                  [unstore_unit]
                        variable= all_units[$i]
                  [/unstore_unit]
            [/then]
      [/if]
{NEXT i}
```

Of course, the first version is much more concise. One should mark :
- Filters are hidden loops[2] fetching all elements of the starting set.
- Criteria composition is more explicit in the second version. We have here an **[and]** tag which is missing in the first one. It shows clearly both condition must be met. In filters, the **[and]** tag is most often implicit, as we already seen.

[2] Internally, it's not always the case, but we've not to deal with internal implementations. Conceptually, filters can always be seen as hidden loops.

At this point, a question arises : where are the starting and result sets we talked about ? They show nowhere. The reply is most often we don't need to see them, because we don't need to create result sets explicitly. We need to use them to specify actions targets or conditions. In the '**modify_unit**' example, we apply the action « change side » to the result set of the filter and then need it no more.
The starting set is implicit too. Most of time, it's the larger available set of objects : i.e. all created units (sometimes including the recall list) or all hexes in the map. In the **moveto** event, it contains only the moving unit.
Once again, we are not telling these sets really exist in Wesnoth engine code. But these are an accurate model of how the filters work in WML.

Result sets and arrays.

A good way to get explicitly the result set is to use the '**store_...**' tags. These actions create arrays containing the result set of the filter they take as parameter. This is very useful in many ways. The common use is of course to store units and locations for some processing or later use. But one can use this to split complex filters and inspect intermediate results. The former '**modify_unit**' example could be written as :

```
[store_unit]
      variable=temp1
      [filter]
            side=2
      [/filter]
[/store_unit]
[store_unit]
      variable=temp2
      [filter]
            find_in=temp1
            [not]
                  race=orc
            [/not]
      [/filter]
[/store_unit]
[modify_unit]
      [filter]
            find_in=temp2
      [/filter]
      side=1
[/modify_unit]
```

This trivial example shows not only how to debug complex filters (inspecting the content of *temp1* and *temp2* arrays), but how to specify a starting set with the '**find_in**' key. Without it, the second '**store_unit**' tag would store all units except orcs. With it, we ask to apply the filter to the content of *temp1* array only (all side 2 units). It's like our card example where we selected the spades first and next the king(s) in the spades stack.
The '**find-in**' key is really precious in many cases : often it's easier to build explicitly an array containing the objects we want to select than creating complex filters to retrieve them. For example, if we want dying units to drop weapons and other units to retrieve them, it can be very difficult to create a filter allowing to select locations where the weapons where dropped. Instead, we can build an array containing their locations (and other informations at

will). Since this array have x and y members, the **find_in** key of a location filter can use it[3]. It would be:

```
# in the die event
[set_variables]
      name=weapons
      mode=append
      [value]
            x=$unit.x
            y=$unit.y
            … anything else, for instance the image name and item id.
      [/value]
[/set_variables]
# drop item, etc…

# and in a moveto event
[event]
      name=moveto
      first_time_only=no
      [filter]
            side=1
            [filter_location]
                  find_in=weapons
            [/filter_location]
      [/filter]
…
```

let's give another example.
The scenario's map features three temples at 10,10 20,20 30,30. We want to give some bonus gold to side 1 if any side 1 unit visits temple 1,2,3 exactly in that order. Here is a solution using result sets arrays :

```
[event]
      name=moveto
      first_time_only=no
      [filter]
            side=1
            x,y=10,10
            [not]
                  find_in=temple_1
            [/not]
      [/filter]
      [store_unit]
            mode=append
            variable=temple_1
            [filter]
                  id=$unit.id
            [/filter]
      [/store_unit]
[/event]
```

In temple_1, we store all side 1 units visiting temple_1, but only once (that's why the **[not] find_in** tag, because units can visit the temple more than once).

[3] It's surprising because this array doesn't contain locations, but it's a feature, not a side effect.

```
[event]
      name=moveto
      first_time_only=no
      [filter]
            side=1
            x,y=20,20
            find_in=temple_1
            [not]
                  find_in=temple_2
            [/not]
      [/filter]
      [store_unit]
            mode=append
            variable=temple_2
            [filter]
                  id=$unit.id
            [/filter]
      [/store_unit]
[/event]
```

This time, we store only units previously stored in temple_1 (= they already visited that temple). Then the last event is obviously :

```
[event]
      name=moveto
      first_time_only=yes
      [filter]
            side=1
            x,y=30,30
            find_in=temple_2
      [/filter]
      [gold]
            side=1
            amount=1000
      [/gold]
      {CLEAR_VARIABLE temple_1,temple_2}
[/event]
```

Ordering and writing

We said earlier that criteria order is not significant. That's right from a theoretical point of view. But, for syntactic reasons and particularly because the radius key in location filters, this is not fully true in WML.
Actually conditions are applied to candidate objects until one proves to be false or all conditions are checked. Then, if some condition proves to be false, remaining conditions are not evaluated (so if there's some radius there, it will not be executed). In some cases, this may be important. One can assume conditions are evaluated in the order they are found except logical operators (and, or, not) which are evaluated after other conditions. It's very important to note that evaluation order of top level conditions (those not embedded in and/or/not tags) is undocumented, which means your code shouldn't rely on it. On the contrary, and/or/not tags are always executed last, in the order they are written.

So in this example:

```
 [filter]
      has_weapon=sword
      side=1
      [or]
            side=2
      [/or]
      gender=female
 [/filter]
```

will be executed using this order:

```
[filter]
      has_weapon=sword
      side=1
      gender=female
      [or]
            side=2
      [/or]
[/filter]
```

One should be aware of this for some reasons:
→ Clarity: your code will be easier to understand and to debug if you avoid meddling conditions, nested filters and logical blocks, and write them in the order they are evaluated.
→ Performance.
Most of time, performance is not an issue. But it can be if you have a lot of units and use **[filter_wml]**. More generally, it's good programming practice to execute the more restrictive test first. Consider this example:

```
[filter]
      race=orc
      x,y=16,22
[/filter]
```

The first condition will be evaluated on all units. But the second one will be evaluated on all orcs. Then if we write:

```
[filter]
      x,y=16,22
      race=orc
 [/filter]
```

obviously, the second condition will be evaluated once at most, and filtering will be faster. (Strictly speaking, I should have wrapped second condition in an **and** tag to force evaluation order).
Remember too that logical operators (**and, or, not**) are not mandatory, but are allowed. So one can use them for clarity sake or to force an evaluation order. It's particularly important when using **or** tags. In the example above one could expect the result set contains all women of sides 1 and 2 wielding a sword. But it contains actually all side 1 women wielding a sword plus all side 2 units. Filters work actually as if top level criteria were enclosed in an implicit **and** tag: so we should read:

```
[filter]
      [and] #implicit
            has_weapon=sword
            side=1
            gender=female
      [/and]
      [or]
            side=2
      [/or]
[/filter]
```

and what we probably wanted is:

```
[filter]
      has_weapon=sword
      gender=female
      [and]
            side=1
            [or]
                  side=2
            [/or]
      [/and]
 [/filter]
```

Note that it could be written:

```
[filter]
      [and]
            has_weapon=sword
            gender=female
      [/and]
      [and]
            side=1
            [or]
                  side=2
            [/or]
      [/and]
 [/filter]
```

This syntax is correct and you can use it if you find it clearer.
Remembering this implicit **and** tag (and writing it explicitly at will) is very important to understand or design complex filters.

Writing complex filters.

Here are some guidelines one can use when writing complex filters. Suppose we want to set up some kind of disease aura harming units standing close to some villages. We shall start writing a sentence describing the feature.
We want to select units who:

Are enemy to side 1
stand on hexes which
are near to
villages
with side 1 units standing on it

Mark we put only one condition on each line to clearly separate them. Mark we sorted them, because some apply to units to be filtered, others to locations, and finally to other (enemy) units standing on locations. Next, we shall add logical operators and parenthesis to clearly specify what we want:

```
Units, who (
    Are enemy to side 1 AND
    stand on locations which (
        (       Are villages AND
                have unit on it who (
                        Belongs to side 1
                )
        ) OR
        are adjacent to THOSE villages radius 3
    )
)
```

Now, we are ready to start building the filter. Since we want units who… it shall be a unit filter. In the standard unit filter, there's no condition directly allowing to state the unit is enemy to side 1. So we have to replace this with something valid in the SUF context. Using parenthesis to avoid errors, we can replace the condition with a side filter because it's valid in unit filters.

```
Units, who (
    (belongs to a side which (
        is enemy to side 1) ) AND
    stand on locations which (
        (       Are villages AND
                have unit on it who (
                        Belongs to side 1
                )
        ) OR
        are adjacent to THOSE villages radius 3
    )
)
```

Now we can write the filter. Here we do it step by step to show how the translation is rather straightforward and how our parenthesis match exactly the subtags.

```
[filter]
      [filter_side]
            [enemy_of]
                  side=1
            [/enemy_of]
      [/filter_side]
            AND
      stand on locations which ( # we start to filter locations there
            (     Are villages AND
                  have unit on it who (
                        Belongs to side 1
                  )
            ) OR
            are adjacent to THOSE villages radius 3
      )
 [/filter]

[filter]
      [filter_side]
            [enemy_of]
                  side=1
            [/enemy_of]
      [/filter_side]
      [and]
            [filter_location]
                  terrain=*^V*
                        AND
                  have unit on it who (# to unit filter again
                        Belongs to side 1
                  )
                  radius=3 # radius is a special case, see below
            [/filter_location]
      [/and]
 [/filter]

[filter]
      [filter_side]
            [enemy_of]
                  side=1
            [/enemy_of]
      [/filter_side]
      [and]
            [filter_location]
                  terrain=*^V*
                  [and]
                        [filter]
                              side=1
                        [/filter]
                  [/and]
                  radius=3 # radius is a special case, see below
            [/filter_location]
      [/and]
 [/filter]
```

Here, we are done. The filter should work as it is, but it looks rather unusual because all these **[and]** blocks. Actually we can delete most of them using a simple rule: **[and]** tags are not needed when they contain one single criterion or a single block, (except if you want to set up an evaluation order). In our example, the **filter_location** block is alone in its **and** tag, and the embedded **filter** as well, so we can avoid those and tags and get finally:

```
[event]
    name=moveto
    first_time_only=no
    [filter]
        [filter_side]
            [enemy_of]
                side=1
            [/enemy_of]
        [/filter_side]
        [filter_location]
            terrain=*^V*
            [filter]
                side=1
            [/filter]
            radius=3
        [/filter_location]
    [/filter]
    [harm_unit]
        [filter]
            id=$unit.id
        [/filter]
        amount=10
        animate=yes
    [/harm_unit]
[/event]
```

Filters uses: events actions and conditions.

Here, we shall deal with filter uses in WML. The language is not fully consistent, mainly to simplify its syntax, so some points can be misleading.

Using in actions

Most actions take a filter as first parameter. The main difficulty here is to know if **[filter]** tags must be used or not. Actually, they're used to avoid confusing keys and criteria when they have the same name[4]. For instance, the **[kill]** action needs a unit filter and has these keys:

*# **animate**: if 'yes', displays the unit dying (fading away).*
*# **fire_event**: if 'yes', triggers any appropriate 'die' events (See EventWML). Note that events are only fired for killed units that have been on the map (as opposed to recall list).*
*# **[secondary_unit]** with a StandardUnitFilter as argument. Do not use a [filter] tag. Has an effect only if fire_event=yes. The first on-map unit matching the filter.*

As we can see, none of these keys and tags are shared with unit filter keys and tags. This means the code parser needs no **[filter]** tag to know which key belongs to the filter and which to the action. But in the code, there's no obvious distinction.

[4] Note there's no specific top level tag for location filters.

```
[kill]
      animate=yes # this belongs to the ‘kill’
      id=BadGuy_101 # this belongs to the unit filter
[/kill]
```

is the correct syntax. Note a common error is to use a **[filter]** tag here. Since this tag is unknown in the context, it is ignored, so the kill action is applied to the starting set, i.e. all units (including the recall lists). Same with the **filter_side** tag which is not needed everywhere (in **store_sides** particularly).
Adversely, **[modify_unit]** obviously requires a **[filter]** tag, because all keys and criteria have the same name:

```
[modify_unit]
      side=2
      side=1
[/modify_unit]
```

This would make no sense of course because one can’t find if side 1 units must be put into side 2 or the contrary. Anyway, there is an exception: the **[store_unit]** tag requires a **[filter]** tag even if there is no ‘**variable**’ key in units description. Another special case is when using terrain action, because the key **terrain** is valid both in location filters and terrain action. Since there is no special tag to delimit location filters, one should write:

```
[terrain]
      [and]
            terrain=Wo* # here is the filter criterion
      [/and]
      terrain=Rr # and the new terrain
[/terrain]
```

The next example can be confusing:

```
[kill]# kill all units standing on deep water
      animate=yes
      [filter_location]
            terrain=Wo
      [/filter_location]
 [/kill]
```

When reading the ‘**kill**’ tag documentation, one will not find any ‘**filter_location**’ or location filter entry. Does this means it’s an undocumented feature ? No. But the ‘**filter_location**’ belongs to the implicit ‘**filter**’ tag of the ‘**kill**’ action and is documented there. It’s kind of:

```
[kill]# kill all units standing on deep water
      animate=yes
      #[filter] we’re filtering units here, not locations
            [filter_location]
                  terrain=Wo
            [/filter_location]
      #[/filter]
 [/kill]
```

Using filters in conditions.

Filters can be used to create conditional expressions. They can be nested in **[have_unit]** or **[have_location]** tags or in nested filters. Here, the result set is not used directly, but its size must fall in the range defined by the '**count**' key. This finally gives a Boolean result: true or false. So one can use them in **[if] [show_if]** conditional actions or in a **[filter_condition]** tag. They are widely used in nested filters too (see the special chapter on this).

Let's give some examples:

```
[have_unit] # this piece of code evaluates to true when no more enemy
leaders are alive
      canrecruit=yes
      [not]
            side=1
      [/not]
      count=0
[/have_unit]

[have_location] # this one is true if at least 5 side 1 units stand on a
village
      terrain=*^V*
      [filter]
            side=1
      [/filter]
      count=5-1000
[/have_location]
```

Note we could also write this condition:

```
[have_unit]
      side=1
      [filter_location]
            terrain=*^V*
      [/filter_location]
      count=5-1000
[/have_unit]
```

Note we could use a **[store_unit]** instead, testing the *length* property of the array:

```
[store_unit]
      variable=temp
      [filter]
            side=1
            [filter_location]
                  terrain=*^V*
            [/filter_location]
      [/filter]
[/store_unit]

#[if] or [filter_condition]
      [variable]
            name=temp.length
            greater_than=4
      [/variable]
```

Using filters in events.

In events, filters are always used in a conditional way, because they state if the event should fire or not. In any event, we can set **[filter_condition]** using **have_unit** or **have_location** (and a more ordinary variable condition, but this is off topic).
Some events use filters in a special way: **moveto** and **attack** events particularly. In them, the filtering apply not on all units as usual, but only on units involved in the event action: one unit only is moving at a time, two units only are involved in a fight. Then the event fires if and only if the involved unit(s) match the filter.
Note that one can use **[filter]** and **[filter_condition]** in the same **moveto** or **attack** event. Both conditions are then ANDed.

Filtering units

Criteria allowed to filter units (in standard unit filters) are listed below. First, the keys dealing with the unit properties (as usual, comma separated lists means conditions are ORed):
id*: can be a comma-separated list, every unit with one of these ids matches.*
type*: can be a list of types*
race*: (Version 1.11 and later only: this can be a comma-separated list)*
ability*: unit has an ability with the given id (not name !)*
side*: the unit is on the given side (can be a list). One can use a [filter_side] instead*
has_weapon*: the unit has a weapon with the given name*
canrecruit*: yes if the unit can recruit (i.e. is a leader)*
gender*: female if the unit is female rather than the default of male*
role*: the unit has been assigned the given role;*
level*: the level of the unit*
defense*: current defense of the unit on current tile*
movement_cost*: current movement cost of the unit on current tile*
x,y*: the position of the unit. (Ranges ? probably because it works in moveto events)*

Not all unit properties are listed here, but they can be used in a **[filter_wml]** sub-tag like this:

```
[filter_wml]
      max_moves=7
[/filter_wml]

[filter_wml]
      [status]
            poisoned=yes
      [/status]
[/filter_wml]
```

Some of them accept comma separated lists or ranges. Some not, but it also possible to use them more than once with **[and]** and **[or]** subtags. For instance :

```
[and]
      race= elf
      [or]
            race= merman
      [/or]
[/and]
```

Other sub tags are dealing with unit relationships:

→ The hex on which they stand: **[filter_location]** which contains a standard location filter.
→ The units adjacent to it: **[filter_adjacent]** which contains another standard unit filter
→ Their visible status relating to a particular side: **[filter_vision]**

These are nested filters ; or in other words, filters used to create conditions and not result sets (see earlier and later).

Custom functions returning a boolean:
formula: FormulaAI like formula.
lua_function: lua function

SUFs accept a **find_in** key too. As we saw earlier, this allows to restrict the starting set to the content of an array.

this_unit

This variable is a special variable defined only inside SUFs. Suppose we want to catch in a filter units at full health. We can use the **hitpoints**, but the problem is we know not which value to use, because every unit type has its own:

```
[filter]
      side=1
      [filter_wml]
            hitpoints=?
      [/filter_wml]
 [/filter]
```

This is why we could have the use of some way to specify the unit being fetched during the filtering.

```
[filter]
      side=1
      [filter_wml]
            hitpoints=$this_unit.max_hitpoints
      [/filter_wml]
 [/filter]
```

This does the trick. The condition value will be updated according to unit properties before executing the check.

Filtering locations

find_in: a location array specifying the starting set.

time_of_day*: one of lawful, chaotic, neutral or liminal.*
time_of_day_id*: one or more from: dawn, morning, afternoon, dusk, first_watch, second_watch, indoors, underground and deep_underground.*

terrain*: comma separated list of terrains.*
x,y*: the same as in the unit filter; supports any range.*
owner_side*: If a valid side number, restricts stored locations to villages belonging to this side. If 0, restricts to all unowned locations (the whole map except villages which belong to some valid side). A hex is considered a village if and only if its [terrain_type] gives_income= parameter was set to yes (which means a side can own that hex).*

[filter_adjacent_location]*: a standard location filter; if present the correct number of adjacent locations must match this filter*

[filter] *with a Standard Unit Filter as argument; if present a unit must also be there*

radius: *this is not strictly speaking a criterion*. It adds to the result set all hexes adjacent to a matching hex and is always applied last, when all criteria are checked. Remember the filtering process is a hidden loop where all candidates are fetched one by one. If a candidate match the filter, radius adds all adjacent hexes (matching the filter or not !). If it don't, it does nothing. This is why this example doesn't work:

```
[filter] # this example doesn't work !
   side=1
   [filter_location]
      x,y=43,32
      radius=5
      [not]
         x,y=43,32
      [/not]
   [/filter_location]
[/filter]
```

The coder here expected the radius action to be performed just after selecting the 43,32 hex, and the **[not]** criterion applied to this hex and it's adjacent radius 5 set. But **radius** is always applied last, even if written before some other conditions. So when using **radius**, a good rule is to create the filter without it at first and to see if it can catch something. Here, it would give:

```
[filter]
   side=1
   [filter_location]
      x,y=43,32
      [not]
         x,y=43,32
      [/not]
   [/filter_location]
[/filter]
```

which is clearly non sense because the two conditions are mutually exclusive.

The solution is to pack the conditions in two different filters:

```
[filter]
   side=1
   [filter_location]
      x,y=43,32
      radius=5
   [/filter_location]
   [and]
      [filter_location]
          [not]
              x,y=43,32
          [/not]
      [/filter_location]
   [/and]
[/filter]
```

or,

```
[filter]
   side=1
   [filter_location]
      [and]
            x,y=43,32
            radius=5
      [/and]
      [not]
            x,y=43,32
      [/not]
   [/filter_location]
[/filter]
```

or, since the x,y keys are defined in **[filter]** too, it can be:

```
[filter]
   side=1
   [filter_location]
            x,y=43,32
      radius=5
   [/filter_location]
   [not]
        x,y=43,32
   [/not]
[/filter]
```

This is why **[filter_radius]** is useful. As we said, radius adds hexes without checking any condition (except proximity of course). If we want to put a condition on hexes added with radius (and them only), we would use it as in next example. Here we want to select forested hexes near villages:

```
   [filter_location]
      terrain=*^V*
      radius=3
      [filter_radius]
            terrain=*^F*
      [/filter_radius]
   [/filter_location]
```

But, this will not work exactly as in our previous example because radius extends outwards from matching locations one step at a time. Only the locations matching the **filter_radius** will be selected AND used to compute the next step. If there's no forest hex near the village, the previous filter will return nothing, even if there are some forest hexes farther in the range.

Note this filter selects the village too ! If we want not, this should be:

```
[filter_location]
   [and]
         terrain=*^V*
         radius=3
         [filter_radius]
               terrain=*^F*
         [/filter_radius]
   [/and]
   [not]
         terrain=*^V*
   [/not]
[/filter_location]
```

Nested filters

Now, we are ready to study how to create nested filters, in other words, filters containing sub filters. In location or unit filters, the documentation says one can insert filters of various kind involving other objects. In this way, we can select unit adjacent to other units or standing on some terrains. Actually, they're not exactly filters: they are conditions or criteria built on filters. In other words, they don't produce a result set but are, like other criteria, expressions evaluating to true or false. That's why many of them have additional keys, like '**count**' (see the filter use in conditions).

We shall discuss this on examples.

Filter_adjacent.

In a unit filter, this allow to create criteria stating which units must be adjacent to the tested unit. In this example, we shall implement an ability named 'escape'. Units having this ability can teleport elsewhere when surrounded by more than 3 enemies. We shall set a **moveto** event to watch the 'surround' event.

```
    [event]
        name=moveto
        first_time_only=no
        [filter]
            # we could set some additional condition on the moving unit
here
            [filter_adjacent]
                ability=escape # this part fetches the adjacent units, not
the moving one.

                is_enemy=yes
            [/filter_adjacent] # this results to true if at least one unit
is found.
        [/filter]
…
    [/event]
```

This will fire each time some enemy unit get close to our able unit. Note we have here not only a standard unit filter (the ability key), but special keys specifying relationships between units : the **is_enemy** key. Another special key is the '**count'** key. It allows to specify how much adjacent units must be found. As far as the default is 1-6, we don't need it here. But this means more than one able unit can be surrounded by a single move.
Now, we must add the 'surrounded' condition. Here, we shall use a new **filter_adjacent** tag, but applying to the able unit (not the moving one):

```
    [event]
        name=moveto
        first_time_only=no
        [filter]
            [filter_adjacent]
                ability=escape # this part filters the adjacent units, not
the moving one.

                [filter_adjacent]
                    is_enemy=yes # this part finds adjacent units to the
able one.

                    count=4-6
                [/filter_adjacent]
                is_enemy=yes
            [/filter_adjacent]
        [/filter]
…
    [/event]
```

Suppose we want now to apply one more condition for the ability to work : able unit must be adjacent to another unit sharing the same ability. We must use a new **filter_adjacent** tag, and since there is already one, use an '**and'** tag to combine them.

```
    [event]
        name=moveto
        first_time_only=no
        [filter]
            # we could set some additional condition here
            [filter_adjacent]
                ability=escape
                [filter_adjacent]
                    is_enemy=yes
                    count=4-6
                [/filter_adjacent]
                [and]
                    [filter_adjacent]
                        ability=escape # these are the needed helpers
                        is_enemy=no
                    [/filter_adjacent]
                [/and]
                is_enemy=yes
            [/filter_adjacent]
        [/filter]
…
    [/event]
```

Note you'll find very few examples of such complex filters in events. Why ? It's because we often need to catch the involved units to take some actions. In our example, we need not only to test if our able units are surrounded but make them teleport as well. As a result, the filtering

process would probably be split in two, moving in the **teleport** filter the code which was in the **filter_adjacent** tag:

```
    [event]
        name=moveto
        first_time_only=no
        [filter]
            # we could set some additional condition here
            [filter_adjacent]
                ability=escape
                is_enemy=yes
                # here was a block …
            [/filter_adjacent]
        [/filter]

        [teleport]
            [filter]
                ability=escape
                [filter_adjacent] # … which was just moved here !
                    is_enemy=yes
                    count=4-6
                [/filter_adjacent]
                [and]
                    [filter_adjacent]
                        ability=escape # these are the needed helpers
                        is_enemy=no
                        #count=1-6 # since it’s the default, we don’t need
this
                    [/filter_adjacent]
                [/and]
            [/filter]
            …
        [/teleport]
    [/event]
```

Filter_location

Filter_location allows to specify on which hex the unit must be standing. Of course, since we already have x,y keys in the standard unit filter, we don’t need to set a filter location for that. Suppose our former ability should work only in forests. The **filter_location** is fitted for that.

```
[filter_location]
      terrain=*^F*
[/filter_location]
```

Where shall we put it ? Certainly not at the first level of the filter: this would make the ability to work if the moving unit (the enemy) stands on forest. So the right place is in level 2 and 3 where we are dealing with the able units.

```
    [event]
        name=moveto
        first_time_only=no
        [filter]
            # we could set some additional condition here
            [filter_adjacent]
                ability=escape
                [filter_adjacent]
                    is_enemy=yes
                    count=4-6
                [/filter_adjacent]
                [and]
                    [filter_adjacent]
                        ability=escape # these are the needed helpers
                        is_enemy=no
                        [filter_location]
                              terrain=*^F*
                        [/filter_location]
                    [/filter_adjacent]
                [/and]
                [filter_location]
                    terrain=*^F*
                [/filter_location]
                is_enemy=yes
            [/filter_adjacent]
        [/filter]
…
    [/event]
```

pitfalls

One thing to avoid designing filters is using redundant or mutually exclusive criteria. In other words, they specify a condition never or always met in any case. Let's give an example:

```
[filter]
      side=1
      [filter_adjacent]
            side=1
            is_enemy=yes
      [/filter_adjacent]
[/filter]
```

It's pretty obvious this filter will always return an empty set because side 1 units can't be enemy to side 1. But this filter is valid and will raise no error ! Now, let's look at this one:

```
[filter]
      side=1
      [filter_adjacent]
            side=2
            is_enemy=yes
      [/filter_adjacent]
[/filter]
```

Here, we can mark the **is_enemy** key is redundant because sides already define if the units are enemy or not. So, if side 1 and 2 are allied, the filter will always return an empty set. If they are not, it will always match, so the **is_enemy** criterion is useless. This filter:

```
[filter]
      side=1
      [filter_adjacent]
            side=2
      [/filter_adjacent]
[/filter]
```

will always return exactly the same result (except if sides configuration is modified during the scenario of course). Another example:

```
[filter]
      type=Horseman,Knight
      [filter_location]
            terrain=M*
      [/filter_location]
[/filter]
```

The result set will always be empty because these units can't walk on mountains. So, there's no need to use such a filter and it's most probably a design error.

Another common error is assuming a filter will always return a single unit or location. In conjunction with **store_unit** or **store_locations**, the result set will be an array or a single variable:

```
[store_unit]
      variable=temp
      [filter]
            … anything
      [/filter]
[/store_unit]

[modify_unit]
      [filter]
            id=$temp.id
      [/filter]
      … something
[/modify_unit]
```

This will work only if the result set contains a single unit. Else, the *temp.id* will be empty, as if the result set was empty. Instead, one should use temp[0].id or temp[$i].id in a loop. Or better in this particular case: **find_in**=temp in the unit filter, because it handles correctly all the cases.

Have fun !

Pyrophorus & Adamant14