ActorWars 2013 Manual

I’ve taken the time to write out a condensed summary of the Actions and Peons action package thingies and other nontrivial miscellaneous junk in the new ActorWars 2013 so that you don’t have to read all the code to be able to create a working entry. I hope it will help.

ActiveActor

An Actor extension that has health, energy, an inventory, and a perform function. The Perform function executes whatever action is passed into it, and subtracts that action’s cost from your energy. If health goes to zero or below zero it removes itself from the grid. If its energy goes below zero, known as deficit spending, health converted to energy less efficiently than Action.siphonselfHP/EP to put energy at 0.

Basic actions

All Actions have a few important abstract methods, perform\* (Contains the code to control the behavior associated with said action), getCost (returns or calculates the cost of this action that is, the amount of energy to be subtracted from the user’s energy), isExclusive (returns whether or not this action Is in the group of actions that only one of can be executed per step). When an action is performed, it executes the code in it’s perform function, and subtracts its getCost from that ActiveActor’s energy. Only one action whose isExclusive evaluates true may be performed per step. Note that a few are left out, like Action.say, because they have no practical use in a match.

\*Note that an Action’s perform is different than an ActiveActor’s.

Here are a list of the premade static Actions, as well as a condensed description of each

* Action.halt()
  + Perform - does nothing. Basically just waits a tick.
  + Cost – 0
  + Is exclusive.
* Action.pass(final Class<?> e, final int num)
  + Perform - If you’re facing another ActiveActor, and you have a “e” in your inventory, “num” of “e” from your inventory and adds it to the inventory of the ActiveActor you are facing. If you have less than “e” it only transfers however many are in your inventory.
  + Cost - 0
  + is not exclusive.
* Action.turn(final int direction)
  + Perform – sets direction to “direction”
  + Cost – 0
  + Is not exclusive
* Action.move()
  + Perform – moves (like a bug) if canMove().
  + Cost – 0
  + Is exclusive
* Action.attackEP(final int energy)
  + Perform – If there is a DestructibleActor in from of the user, it attacks with the energy specified. Damage done = ((“energy” + 4)1/2) – 2
  + Cost – “energy”
  + Is exclusive
* Action.attackHP(final int health)
  + Perform - If there is a DestructibleActor in front of the user, it attacks dealing “health” damage.
  + Cost – ((“health” + 2)2) – 4
  + Is exclusive
* Action.heal(final int scalar)
  + Perform – If there is a DestructbleActor in front of the user, it heals it for “scalar”
  + Cost – (5 + (5 \* “scalar”)) \* “scalar”
  + Is exclusive
* Action.siphonSelfHP(final int health)
  + Perform – Subtracts “health” from your health, and raises your energy by (((6\*”health”)7)2-48)/12
  + Cost – 0
  + Is exclusive
* Action.siphonSelfEP(final int energy)
  + Perform –does same thing as Action.siphonselfHP, but passes energy instead of health
  + Cost – 0
  + Is exclusive
* Action.healSelf(final int energy)
  + Perform – Heals the actor for ((((12\*”energy”)+49)1/2)-7)/6 health
  + Cost – “energy”
  + Is exclusive
* Action.place(final Class<?> e)
  + Perform – Puts an “e” from your inventory onto the grid in front of you.
  + Cost – 0
  + Is exclusive
* Action.push()
  + Perform – moves a Movable in front of you forward, and then you forward.
  + Cost – 10
  + Is exclusive
* Action.use(final Class<?> e)
  + Perform – uses one of “e” in your inventory. Note that different classes have different uses.
  + Cost – 0
  + Is exclusive
* Action.craft(final Class<?> e)
  + Perform – Puts an “e” in your inventory iff you have the things in its recipe. Those things are then removed from your inventory.
  + Cost – 0
  + Is not exclusive
* Action.spawn(final Class<?> e)
  + Perform – Puts a new “e” in the grid.
  + Cost – 400 (Note: you can’t deficit spend on spawning.)
  + Is exclusive
* Action.toggle(Final ModifiableBoolean value)
  + Perform – Inverts the given ModifiableBoolean.
  + Cost – 0
  + Is not exclusive
* Action.set(Final ModifiableBoolean value, Final ModifiableBoolean newvalue)
  + Perfrom – Sets “value” to “newvalue”
  + Cost – 0
  + Is not exclusive
* Action.increment(final ModifiableInteger value)
  + Increments “value”
  + Cost – 0
  + Is not exclusive
* Action.decrement(final ModifiableInteger value)
  + decrements “value”
  + Cost – 0
  + Is not exclusive
* Action.set(final ModifiableInteger value, final ModifiableInteger newvalue)
  + sets “value” to “newvalue”
  + Cost – 0
  + Is not exclusive

Peons

Technically you could extend ActiveActor to create your entry; it has all that is required, health, energy, an inventory, and a perform function to use actions. But the Peon has an added utility, the Action queue. The Peon class has an ArrayList of Actions, the myactions queue. Every call to peonAct passes all Actions in myactions up to and including the next exclusive action from position 0 into the Peons perform function. This allows Peons to plan ahead, but also adds the overhead of dealing with that queue. Peons also have a few Actions bundle thingies. These sort of decompress dynamically into a bunch of regular action, or perform more complex Actions, such as pathfinding.

Here they are.

* Peon.count(final int c)
  + Perform – Decompresses into “c” halts.
  + Cost – 0
  + Is exclusive
* Peon.moveToGradual(final ModifiableLocation location)
  + Perform – Decompresses into a set of moves and turns that will get you to “location” or it’s closest empty adjacent if its occupied.
  + Cost – 0
  + Is not exclusive, but decompresses into things that are
* Peon.repeat(final Action myact, final ModifiableBoolean b)
  + Perform – Performs “myact” until !“b”
  + Cost – 0, but could decompress into something with nonzero cost
  + Is not exclusive, but could decompress into something that is.
* Peon.conditionalAct(final ModifiableBoolean b, final Action ifaction)
  + Perform – Performs “ifaction” if “b”
  + Cost – 0, but could bring about something with nonzero cost.
  + Is not exclusive, but could bring about something exclusive.
* Peon.conditionalAct(final ModifiableBoolean b, final Action ifaction, final Action elseaction)
  + Perform – Performs “ifaction” if “b” else “elseaction
  + Cost – 0, but could bring about something with nonzero cost.
  + Is not exclusive, but could bring about something exclusive.
* Peon.placeAt(final Class<?> e, final Location l)
  + Perform – goes and places “e” at “l”
  + Cost – 0
  + Is not exclusive, but decompresses into things that are.

List of movables: Rock.

List of Placeables: Fence, Rock, Wheat, Wood

List of Usables (See class for use details):Pickaxe, Axe, Wheat

List of Craftables(see class for recipe):Pickaxe, Axe, fence