目录

Behaviourtree

```
require("class")

SUCCESS = "SUCCESS"
FAILED = "FAILED"
READY = "READY"
RUNNING = "RUNNING"

--------------------------------------------------------------------------------

BT = Class(function(self, inst, root)
    self.inst = inst
    self.root = root
end)

function BT:ForceUpdate()
```

```
    self.forceupdate = true
end
function BT:Update()

    self.root:Visit()
    self.root:SaveStatus()
    self.root:Step()

    self.forceupdate = false
end

function BT:Reset()
    self.root:Reset()
end

function BT:Stop()
    self.root:Stop()
end

function BT:GetSleepTime()
    if self.forceupdate then
        return 0
    end

    return self.root:GetTreeSleepTime()
end

function BT:__tostring()
    return self.root:GetTreeString()
end



--------------------------------------------------------------------------------

BehaviourNode = Class(function (self, name, children)
    self.name = name or ""
    self.children = children
    self.status = READY
    self.lastresult = READY
    if children then
        for i,k in pairs(children) do
            k.parent = self
        end
    end
end)

function BehaviourNode:DoToParents(fn)
    if self.parent then
        fn(self.parent)
```

```
        return self.parent:DoToParents(fn)
    end
end

function BehaviourNode:GetTreeString(indent)
    indent = indent or ""
    local str = string.format("%s%s>%2.2f\n", indent, self:GetString(),
self:GetTreeSleepTime() or 0)
    if self.children then
        for k, v in ipairs(self.children) do
            str = str .. v:GetTreeString(indent .. "   >")
        end
    end
    return str
end

function BehaviourNode:DBString()
    return ""
end

function BehaviourNode:Sleep(t)
    self.nextupdatetime = GetTime() + t
end

function BehaviourNode:GetSleepTime()

    if self.status == RUNNING and not self.children and not self:is_a(ConditionNode)
then
        if self.nextupdatetime then
            local time_to = self.nextupdatetime - GetTime()
            if time_to < 0 then
                time_to = 0
            end
            return time_to
        end
        return 0
    end

    return nil
end

function BehaviourNode:GetTreeSleepTime()

    local sleeptime = nil
    if self.children then
        for k,v in ipairs(self.children) do
            if v.status == RUNNING then
                local t = v:GetTreeSleepTime()
                if t and (not sleeptime or sleeptime > t) then
```

```
                    sleeptime = t
                end
            end
        end
    end

    local my_t = self:GetSleepTime()

    if my_t and (not sleeptime or sleeptime > my_t) then
        sleeptime = my_t
    end

    return sleeptime
end

function BehaviourNode:GetString()
    local str = ""
    if self.status == RUNNING then
        str = self:DBString()
    end
    return string.format([[%s - %s <%s> (%s)]], self.name, self.status or "UNKNOWN",
self.lastresult or "?", str)
end

function BehaviourNode:Visit()
    self.status = FAILED
end

function BehaviourNode:SaveStatus()
    self.lastresult = self.status
    if self.children then
        for k,v in pairs(self.children) do
            v:SaveStatus()
        end
    end
end

function BehaviourNode:Step()
    if self.status ~= RUNNING then
        self:Reset()
    elseif self.children then
        for k, v in ipairs(self.children) do
            v:Step()
        end
    end
end

function BehaviourNode:Reset()
    if self.status ~= READY then
```

```
        self.status = READY
        if self.children then
            for idx, child in ipairs(self.children) do
                child:Reset()
            end
        end
    end
end

function BehaviourNode:Stop()
    if self.OnStop then
        self:OnStop()
    end
    if self.children then
        for idx, child in ipairs(self.children) do
            child:Stop()
        end
    end
end


--------------------------------------------------------------------------------
DecoratorNode = Class(BehaviourNode, function(self, name, child)
  BehaviourNode._ctor(self, name or "Decorator", {child})
end)

--------------------------------------------------------------------------------


ConditionNode = Class(BehaviourNode, function(self, fn, name)
    BehaviourNode._ctor(self, name or "Condition")
    self.fn = fn
end)

function ConditionNode:Visit()
    if self.fn() then
        self.status = SUCCESS
    else
        self.status = FAILED
    end
end


--------------------------------------------------------------------------------


ConditionWaitNode = Class(BehaviourNode, function(self, fn, name)
    BehaviourNode._ctor(self, name or "Wait")
    self.fn = fn
```

```
end)

function ConditionWaitNode:Visit()
    if self.fn() then
        self.status = SUCCESS
    else
        self.status = RUNNING
    end
end


--------------------------------------------------------------------------------


ActionNode = Class(BehaviourNode, function(self, action, name)
    BehaviourNode._ctor(self, name or "ActionNode")
    self.action = action
end)

function ActionNode:Visit()
    self.action()
    self.status = SUCCESS
end


--------------------------------------------------------------------------------


WaitNode = Class(BehaviourNode, function(self, time)
    BehaviourNode._ctor(self, "Wait")
    self.wait_time = time
end)

function WaitNode:DBString()
    local w = self.wake_time - GetTime()
    return string.format("%2.2f", w)
end

function WaitNode:Visit()
    local current_time = GetTime()

    if self.status ~= RUNNING then
        self.wake_time = current_time + self.wait_time
        self.status = RUNNING
    end

    if self.status == RUNNING then
        if current_time >= self.wake_time then
            self.status = SUCCESS
```

```
        else
            self:Sleep(current_time - self.wake_time)
        end
    end

end


--------------------------------------------------------------------------------

SequenceNode = Class(BehaviourNode, function(self, children)
    BehaviourNode._ctor(self, "Sequence", children)
    self.idx = 1
end)

function SequenceNode:DBString()
    return tostring(self.idx)
end


function SequenceNode:Reset()
    self._base.Reset(self)
    self.idx = 1
end

function SequenceNode:Visit()

    if self.status ~= RUNNING then
        self.idx = 1
    end

    local done = false
    while self.idx <= #self.children do

        local child = self.children[self.idx]
        child:Visit()
        if child.status == RUNNING or child.status == FAILED then
            self.status = child.status
            return
        end

        self.idx = self.idx + 1
    end

    self.status = SUCCESS
end

--------------------------------------------------------------------------------
```

```
SelectorNode = Class(BehaviourNode, function(self, children)
    BehaviourNode._ctor(self, "Selector", children)
    self.idx = 1
end)

function SelectorNode:DBString()
    return tostring(self.idx)
end


function SelectorNode:Reset()
    self._base.Reset(self)
    self.idx = 1
end

function SelectorNode:Visit()

    if self.status ~= RUNNING then
        self.idx = 1
    end

    local done = false
    while self.idx <= #self.children do

        local child = self.children[self.idx]
        child:Visit()
        if child.status == RUNNING or child.status == SUCCESS then
            self.status = child.status
            return
        end

        self.idx = self.idx + 1
    end

    self.status = FAILED
end
--------------------------------------------------------------------------------

NotDecorator = Class(DecoratorNode, function(self, child)
    DecoratorNode._ctor(self, "Not", child)
end)

function NotDecorator:Visit()
  local child = self.children[1]
  child:Visit()
  if child.status == SUCCESS then
    self.status = FAILED
  elseif child.status == FAILED then
    self.status = SUCCESS
```

```
  else
    self.status = child.status
  end
end
-------------------------------------------------------------------------------

FailIfRunningDecorator = Class(DecoratorNode, function(self, child)
    DecoratorNode._ctor(self, "FailIfRunning", child)
end)

function FailIfRunningDecorator:Visit()
  local child = self.children[1]
  child:Visit()
  if child.status == RUNNING then
    self.status = FAILED
  else
    self.status = child.status
  end
end

-------------------------------------------------------------------------------


LoopNode = Class(BehaviourNode, function(self, children, maxreps)
    BehaviourNode._ctor(self, "Sequence", children)
    self.idx = 1
    self.maxreps = maxreps
    self.rep = 0
end)

function LoopNode:DBString()
    return tostring(self.idx)
end


function LoopNode:Reset()
    self._base.Reset(self)
    self.idx = 1
    self.rep = 0
end

function LoopNode:Visit()

    if self.status ~= RUNNING then
        self.idx = 1
        self.rep = 0
    end

    local done = false
```

```
    while self.idx <= #self.children do

        local child = self.children[self.idx]
        child:Visit()
        if child.status == RUNNING or child.status == FAILED then
            if child.status == FAILED then
                --print("EXIT LOOP ON FAIL")
            end
            self.status = child.status
            return
        end

        self.idx = self.idx + 1
    end

    self.idx = 1

    self.rep = self.rep + 1
    if self.maxreps and self.rep >= self.maxreps then
        --print("DONE LOOP")
        self.status = SUCCESS
    else
        for k,v in ipairs(self.children) do
            v:Reset()
        end

    end
end

--------------------------------------------------------------------------------

RandomNode = Class(BehaviourNode, function(self, children)
    BehaviourNode._ctor(self, "Random", children)
end)


function RandomNode:Reset()
    self._base.Reset(self)
    self.idx = nil
end


function RandomNode:Visit()

    local done = false

    if self.status == READY then
        --pick a new child
        self.idx = math.random(#self.children)
```

```
        local start = inst.idx
        while true do

            local child = self.children[self.idx]
            child:Visit()

            if child.status ~= FAILED then
                self.status = child.status
                return
            end

            self.idx = self.idx + 1
            if self.idx == #self.children then
                self.idx = 1
            end

            if self.idx == start then
                inst.status = FAILED
                return
            end
        end

    else
        local child = self.children[self.idx]
        child:Visit()
        self.status = child.status
    end

end

--------------------------------------------------------------------------------


PriorityNode = Class(BehaviourNode, function(self, children, period)
    BehaviourNode._ctor(self, "Priority", children)
    self.period = period or 1
end)

function PriorityNode:GetSleepTime()
    if self.status == RUNNING then

        if not self.period then
            return 0
        end


        local time_to = 0
        if self.lasttime then
            time_to = self.lasttime + self.period - GetTime()
```

```
            if time_to < 0 then
                time_to = 0
            end
        end

        return time_to
    elseif self.status == READY then
        return 0
    end

    return nil

end


function PriorityNode:DBString()
    local time_till = 0
    if self.period then
       time_till = (self.lasttime or 0) + self.period - GetTime()
    end

    return string.format("execute %d, eval in %2.2f", self.idx or -1, time_till)
end


function PriorityNode:Reset()
    self._base.Reset(self)
    self.idx = nil
end

function PriorityNode:Visit()

    local time = GetTime()
    local do_eval = not self.lasttime or not self.period or self.lasttime + self.period
< time
    local oldidx = self.idx


    if do_eval then

        local old_event = nil
        if self.idx and self.children[self.idx]:is_a(EventNode) then
            old_event = self.children[self.idx]
        end

        self.lasttime = time
        local found = false
        for idx, child in ipairs(self.children) do
```

```
            local should_test_anyway = old_event and child:is_a(EventNode) and
old_event.priority <= child.priority
            if not found or should_test_anyway then

                if child.status == FAILED or child.status == SUCCESS then
                    child:Reset()
                end
                child:Visit()
                local cs = child.status
                if cs == SUCCESS or cs == RUNNING then
                    if should_test_anyway and self.idx ~= idx then
                        self.children[self.idx]:Reset()
                    end
                    self.status = cs
                    found = true
                    self.idx = idx
                end
            else

                child:Reset()
            end
        end
        if not found then
            self.status = FAILED
        end

    else
        if self.idx then
            local child = self.children[self.idx]
            if child.status == RUNNING then
                child:Visit()
                self.status = child.status
                if self.status ~= RUNNING then
                    self.lasttime = nil
                end
            end
        end
    end

end


--------------------------------------------------------------------------------


ParallelNode = Class(BehaviourNode, function(self, children, name)
    BehaviourNode._ctor(self, name or "Parallel", children)
end)
```

```
function ParallelNode:Step()
    if self.status ~= RUNNING then
        self:Reset()
    elseif self.children then
        for k, v in ipairs(self.children) do
            if v.status == SUCCESS and v:is_a(ConditionNode) then
                v:Reset()
            end
        end
    end
end

function ParallelNode:Visit()
    local done = true
    local any_done = false
    for idx, child in ipairs(self.children) do

        if child:is_a(ConditionNode) then
            child:Reset()
        end

        if child.status ~= SUCCESS then
            child:Visit()
            if child.status == FAILED then
                self.status = FAILED
                return
            end
        end

        if child.status == RUNNING then
            done = false
        else
            any_done = true
        end


    end

    if done or (self.stoponanycomplete and any_done) then
        self.status = SUCCESS
    else
        self.status = RUNNING
    end
end

ParallelNodeAny = Class(ParallelNode, function(self, children)
    ParallelNode._ctor(self, children, "Parallel(Any)")
    self.stoponanycomplete = true
```

```
end)



--------------------------------------------------------------------------------


EventNode = Class(BehaviourNode, function(self, inst, event, child, priority)
    BehaviourNode._ctor(self, "Event("..event..")", {child})
    self.inst = inst
    self.event = event
    self.priority = priority or 0

    self.eventfn = function(inst, data) self:OnEvent(data) end
    self.inst:ListenForEvent(self.event, self.eventfn)
    --print(self.inst, "EventNode()", self.event)
end)

function EventNode:OnStop()
    --print(self.inst, "EventNode:OnStop()", self.event)
    if self.eventfn then
        self.inst:RemoveEventCallback(self.event, self.eventfn)
        self.eventfn = nil
    end
end

function EventNode:OnEvent(data)
    --print(self.inst, "EventNode:OnEvent()", self.event)

    if self.status == RUNNING then
        self.children[1]:Reset()
    end
    self.triggered = true
    self.data = data

    if self.inst.brain then
        self.inst.brain:ForceUpdate()
    end

    self:DoToParents(function(node) if node:is_a(PriorityNode) then node.lasttime = nil
end end)

    --wake the parent!
end

function EventNode:Step()
    self._base.Step(self)
    self.triggered = false
end
```

```
function EventNode:Reset()
    self.triggered = false
    self._base.Reset(self)
end

function EventNode:Visit()

    if self.status == READY and self.triggered then
        self.status = RUNNING
    end

    if self.status == RUNNING then
        if self.children and #self.children == 1 then
            local child = self.children[1]
            child:Visit()
            self.status = child.status
        else
            self.status = FAILED
        end
    end

end

--------------------------------------------------------------

function WhileNode(cond, name, node)
    return ParallelNode
        {
            ConditionNode( cond, name),
            node
        }
end

--------------------------------------------------------------

function IfNode(cond, name, node)
    return SequenceNode
        {
            ConditionNode( cond, name),
            node
        }
end
--------------------------------------------------------------
```

DoAction

```
DoAction = Class(BehaviourNode, function(self, inst, getactionfn, name, run)
    BehaviourNode._ctor(self, name or "DoAction")
    self.inst = inst
    self.shouldrun = run
    self.action = nil
    self.getactionfn = getactionfn
end)

function DoAction:OnFail()
    self.pendingstatus = FAILED
end

function DoAction:OnSucceed()
    self.pendingstatus = SUCCESS
end


function DoAction:Visit()

    if self.status == READY then
        local action = self.getactionfn(self.inst)

        if action then
            action:AddFailAction(function() self:OnFail() end)
            action:AddSuccessAction(function() self:OnSucceed() end)
            self.pendingstatus = nil
            self.inst.components.locomotor:PushAction(action, self.shouldrun)
            self.action = action;
            self.status = RUNNING
        else
            self.status = FAILED
        end
    end

    if self.status == RUNNING then
        if self.pendingstatus then
            self.status = self.pendingstatus
        elseif not self.action:IsValid() then
            self.status = FAILED
        end
    end

end
```

AttackWall

```
AttackWall = Class(BehaviourNode, function(self, inst)
    BehaviourNode._ctor(self, "AttackWall")
    self.inst = inst
end)

function AttackWall:__tostring()
    return string.format("target %s", tostring(self.target))
end

function AttackWall:Visit()

    if self.status == READY then

        local radius = 1.5 + (self.inst.Physics and self.inst.Physics:GetRadius() or 0)
    self.target = FindEntity(self.inst, radius,
      function(guy)
        if guy:HasTag("wall") and self.inst.components.combat:CanTarget(guy) then
          local angle = anglediff(self.inst.Transform:GetRotation(),
self.inst:GetAngleToPoint(Vector3(guy.Transform:GetWorldPosition() )))
          return math.abs(angle) < 30
        end

      end)

    if self.target then
      self.status = RUNNING
      self.inst.components.locomotor:Stop()
      self.done = false
    else
      self.status = FAILED
    end

    end

    if self.status == RUNNING then
        --local is_attacking = self.inst.sg:HasStateTag("attack")
        if not self.target or not self.target.entity:IsValid() or
(self.target.components.health and self.target.components.health:IsDead())then
            self.status = FAILED
            self.inst.components.locomotor:Stop()
        else
      if self.inst.components.combat:TryAttack(self.target) then
        self.status = SUCCESS
      else
        self.status = FAILED
      end
      self:Sleep(1)
```

```
        end
    end
end
```

AvoidLight

```
AvoidLight = Class(BehaviourNode, function(self, inst)
    BehaviourNode._ctor(self, "AvoidLight")
    self.inst = inst
    self.waiting = false
    self.phasechangetime = 0
end)



function AvoidLight:Wait(t)
    self.waittime = t+GetTime()
    self:Sleep(t)
end


function AvoidLight:PickNewAngle()

    local angles = {}

    if self.inst.Physics:CheckGridOffset(0,-1) then table.insert(angles, -90) end
    if self.inst.Physics:CheckGridOffset(0,1) then table.insert(angles, 90) end
    if self.inst.Physics:CheckGridOffset(-1,0) then table.insert(angles, 180) end
    if self.inst.Physics:CheckGridOffset(1,0) then table.insert(angles, 0) end

    local angle = 0

    local light = self.inst.LightWatcher:GetLightAngle()
    if light then
        table.sort(angles, function(a,b) return anglediff(a, light) <
anglediff(b,light) end)
        angle = angles[1]
    else
        angle = angles[math.random(#angles)]
    end

    angle = angle + math.random()*90-45
    return angle
end

function AvoidLight:Visit()
```

```
    if self.status == READY then
        self.status = RUNNING
        --self.inst.Steering:SetActive(true)
    end

    if self.status == RUNNING then
        local in_light = self.inst.LightWatcher:IsInLight()

        local t = GetTime()
        if t > self.phasechangetime or (self.waiting and in_light) then

            self.waiting = not self.waiting

            if self.waiting then
                self.phasechangetime = .2+math.random()*.25
                self.inst.components.locomotor:Stop()
            else
                self.angle = self:PickNewAngle()
                self.phasechangetime = t + 1+math.random()*3
            end

        end

        if not self.waiting then

            local light = self.inst.LightWatcher:GetLightAngle()
            if light then

                self.inst.entity:LocalToWorldSpace(1,0,0)

                self.angle = light + 180 + math.random()*60-30
            end
            self.inst.components.locomotor:WalkInDirection(self.angle)
            self:Wait(.1)
        end
    end
end
```

ChaseAndAttack

```
ChaseAndAttack = Class(BehaviourNode, function(self, inst, max_chase_time,
give_up_dist, max_attacks, findnewtargetfn)
    BehaviourNode._ctor(self, "ChaseAndAttack")
    self.inst = inst
    self.findnewtargetfn = findnewtargetfn
    self.max_chase_time = max_chase_time
    self.give_up_dist = give_up_dist
```

```
    self.max_attacks = max_attacks
    self.numattacks = 0

    -- we need to store this function as a key to use to remove itself later
    self.onattackfn = function(inst, data)
        self:OnAttackOther(data.target)
    end

    self.inst:ListenForEvent("onattackother", self.onattackfn)
    self.inst:ListenForEvent("onmissother", self.onattackfn)
end)

function ChaseAndAttack:__tostring()
    return string.format("target %s", tostring(self.inst.components.combat.target))
end

function ChaseAndAttack:OnStop()
    self.inst:RemoveEventCallback("onattackother", self.onattackfn)
    self.inst:RemoveEventCallback("onmissother", self.onattackfn)
end

function ChaseAndAttack:OnAttackOther(target)
    --print ("on attack other", target)
    self.numattacks = self.numattacks + 1
    self.startruntime = nil -- reset max chase time timer
end

function ChaseAndAttack:Visit()

    local combat = self.inst.components.combat
    if self.status == READY then

        combat:ValidateTarget()

        if not combat.target and self.findnewtargetfn then
            combat.target = self.findnewtargetfn(self.inst)
        end


        if combat.target then
            self.inst.components.combat:BattleCry()
            self.startruntime = GetTime()
            self.numattacks = 0
            self.status = RUNNING
        else
            self.status = FAILED
        end

    end
```

```
    if self.status == RUNNING then

       local is_attacking = self.inst.sg:HasStateTag("attack")

        if not combat.target or not combat.target.entity:IsValid() then
            self.status = FAILED
            combat:SetTarget(nil)
            self.inst.components.locomotor:Stop()
        elseif combat.target.components.health and
combat.target.components.health:IsDead() then
            self.status = SUCCESS
            combat:SetTarget(nil)
            self.inst.components.locomotor:Stop()
        else


            local hp = Point(combat.target.Transform:GetWorldPosition())
            local pt = Point(self.inst.Transform:GetWorldPosition())
            local dsq = distsq(hp, pt)
            local angle = self.inst:GetAngleToPoint(hp)
            local r= self.inst.Physics:GetRadius()+ combat.target.Physics:GetRadius() +
.1
            local running = self.inst.components.locomotor:WantsToRun()

            if (running and dsq > r*r) or (not running and dsq >
combat:CalcAttackRangeSq() ) then
                --self.inst.components.locomotor:RunInDirection(angle)
                self.inst.components.locomotor:GoToPoint(hp, nil, true)
            elseif not (self.inst.sg and self.inst.sg:HasStateTag("jumping")) then
                self.inst.components.locomotor:Stop()
                if self.inst.sg:HasStateTag("canrotate") then
                    self.inst:FacePoint(hp)
                end
            end

            if combat:TryAttack() then
                -- reset chase timer when attack hits, not on attempts
            else
                if not self.startruntime then
                    self.startruntime = GetTime()
                    self.inst.components.combat:BattleCry()
                end
            end


            if self.max_attacks and self.numattacks >= self.max_attacks then
                self.status = SUCCESS
                self.inst.components.combat:SetTarget(nil)
```

```
                self.inst.components.locomotor:Stop()
                return
            end

            if self.give_up_dist then
                if dsq >= self.give_up_dist*self.give_up_dist then
                    self.status = FAILED
                    self.inst.components.combat:GiveUp()
                    self.inst.components.locomotor:Stop()
                    return
                end
            end

            if self.max_chase_time and self.startruntime then
                local time_running = GetTime() - self.startruntime
                if time_running > self.max_chase_time then
                    self.status = FAILED
                    self.inst.components.combat:GiveUp()
                    self.inst.components.locomotor:Stop()
                    return
                end
            end
            self:Sleep(.125)

        end

    end
end
```

ChattyNode

```
ChattyNode = Class(BehaviourNode, function(self, inst, chatlines, child)
    BehaviourNode._ctor(self, "ChattyNode", {child})

    self.inst = inst
    self.chatlines = chatlines
    self.nextchattime = nil
end)


function ChattyNode:Visit()
    local child = self.children[1]

    child:Visit()
    self.status = child.status

    if self.status == RUNNING then
```

```
        local t = GetTime()

        if not self.nextchattime or t > self.nextchattime then

            local str = self.chatlines[math.random(#self.chatlines)]
            self.inst.components.talker:Say(str)
            self.nextchattime = t + 10 +math.random()*10
        end
        if self.nextchattime then
            self:Sleep(self.nextchattime - t)
        end
    end

end
```

ControlMinions

```
ControlMinions = Class(BehaviourNode, function(self, inst)
   BehaviourNode._ctor(self, "ControlMinions")
   self.inst = inst
   self.ms = inst.components.minionspawner
   self.radius = nil
   self.minionrange = 3.5
end)

function ControlMinions:GetClosestMinion(item, minions)
   local pt = item:GetPosition()
   local inrange = {}
   for k,v in pairs(minions) do
      if v ~= item then
         local dist = math.sqrt(distsq(pt, v:GetPosition()))
         if dist <= self.minionrange then
            table.insert(inrange, {mn = v, distance = dist})
         end
      end
   end

   if #inrange > 0 then
      table.sort(inrange, function(a,b) return (a.distance) < (b.distance) end)
      return inrange[1].mn
   end
end

function ControlMinions:CanActOn(item)
   return item:IsOnValidGround() and
   item:GetTimeAlive() > 1 and
   item:IsValid() and not
```

```
   item:HasTag("irreplaceable") and not
   (item:HasTag("lureplant") or item:HasTag("eyeplant") or item:HasTag("notarget")) and
not
   (item.components.inventoryitem and (item.components.container or
item.components.inventoryitem:IsHeld())) and not
   (item.components.pickable and not (item.components.pickable:CanBePicked() or
item.components.pickable.caninteractwith))
end


function ControlMinions:Visit()
   local minions = {}
   if self.status == READY then
      if self.ms.numminions > 0 then
         self.status = RUNNING
      else
         self.status = FAILED
      end
   end

   if self.status == RUNNING then

      if not self.radius then    --Get the distance you need to look for things within.
         if self.ms.minionpositions then
            local rad = math.sqrt(distsq(self.inst:GetPosition(),
self.ms.minionpositions[#self.ms.minionpositions]))
            self.radius = rad + (rad * 0.1)
         end
      end

      if not self.radius then
         self.status = FAILED
         return
      end

      local pt = self.inst:GetPosition()
      local ents = nil
      if pt then
         ents = TheSim:FindEntities(pt.x, pt.y, pt.z, self.radius)  --find all entities
within required radius
      end
      if ents and #ents > 0 then
         for k,v in pairs(ents) do
            if self:CanActOn(v) then
               local mn = self:GetClosestMinion(v, self.ms.minions)
               if mn and not mn.sg:HasStateTag("busy") then
                  if (v.components.crop and v.components.crop:IsReadyForHarvest()) or
                     (v.components.stewer and v.components.stewer.done) or
                        (v.components.dryer and v.components.dryer:IsDone()) then
```

```
                        --Harvest!
                        local ba = BufferedAction(mn,v,ACTIONS.HARVEST)
                       ba.distance = 4
                       mn:PushBufferedAction(ba)
                      elseif (v.components.pickable and
v.components.pickable:CanBePicked() and v.components.pickable.caninteractwith) then
                        --Pick!
                       local ba = BufferedAction(mn,v,ACTIONS.PICK)
                       ba.distance = 4
                       mn:PushBufferedAction(ba)
                      elseif (v.components.inventoryitem and
v.components.inventoryitem.cangoincontainer and not v.components.container and not
v.components.inventoryitem:IsHeld()) then
                        --Pick up!
                        local ba = BufferedAction(mn,v,ACTIONS.PICKUP)
                       ba.distance = 4
                       mn:PushBufferedAction(ba)
                      end
                      local ba = mn:GetBufferedAction()

                      if ba then
                         mn:FacePoint(Vector3(ba.target.Transform:GetWorldPosition()),
true)
                          end

               end
            end
         end
         self.status = SUCCESS
      else
         self.status = FAILED
      end

   end

end
```

FaceEntity

```
FaceEntity = Class(BehaviourNode, function(self, inst, getfn, keepfn, timeout)
    BehaviourNode._ctor(self, "FaceEntity")
    self.inst = inst
    self.getfn = getfn
    self.keepfn = keepfn

    self.timeout = timeout
    self.starttime = nil
```

```
end)


function FaceEntity:Visit()

    if self.status == READY then
        self.target = self.getfn(self.inst)

        if self.target then
            self.status = RUNNING
            self.inst.components.locomotor:Stop()
            self.starttime = GetTime()
        else
            self.status = FAILED
        end

    end

    if self.status == RUNNING then

        --uhhhh....
        if self.inst.sg:HasStateTag("idle") and self.inst.sg.currentstate.name ~=
"alert" and self.inst.sg.sg.states.alert then
            self.inst.sg:GoToState("alert")
        end

        if self.timeout and self.starttime then
            local totaltime = GetTime() - self.starttime
            if totaltime > self.timeout then
                self.status = SUCCESS
                return
            end
        end

        if self.keepfn(self.inst, self.target) then
            if self.inst.sg:HasStateTag("canrotate") then
                self.inst:FacePoint(Point(self.target.Transform:GetWorldPosition()))
            end

        else
            self.status = FAILED
        end
        self:Sleep(.5)
    end

end
```

FindFlower

```
local SEE_DIST = 30

FindFlower = Class(BehaviourNode, function(self, inst)
    BehaviourNode._ctor(self, "FindFlower")
    self.inst = inst
end)

function FindFlower:DBString()
    return string.format("Go to flower %s",
tostring(self.inst.components.pollinator.target))
end

function FindFlower:Visit()

    if self.status == READY then
        self:PickTarget()
        if self.inst.components.pollinator and self.inst.components.pollinator.target
then
         local action = BufferedAction(self.inst,
self.inst.components.pollinator.target, ACTIONS.POLLINATE, nil, nil, nil, 0.1)
         self.inst.components.locomotor:PushAction(action, self.shouldrun)
         self.status = RUNNING
      else
         self.status = FAILED
        end
    end

    if self.status == RUNNING then
        if not self.inst.components.pollinator.target
           or not
self.inst.components.pollinator:CanPollinate(self.inst.components.pollinator.target)
           or FindEntity(self.inst.components.pollinator.target, 2, function(guy)
return guy ~= self.inst and guy.components.pollinator and
guy.components.pollinator.target == self.inst.components.pollinator.target end) then
            self.status = FAILED
        end
    end
end

function FindFlower:PickTarget()
    local closestFlower = GetClosestInstWithTag("flower", self.inst, SEE_DIST)
    if closestFlower
      and self.inst.components.pollinator
      and self.inst.components.pollinator:CanPollinate(closestFlower)
      and not FindEntity(closestFlower, 2, function(guy) return
guy.components.pollinator and guy.components.pollinator.target == closestFlower end)
then
```

```
        self.inst.components.pollinator.target = closestFlower
    else
        self.inst.components.pollinator.target = nil
    end
end
```

FindLight

```
local SEE_DIST = 30
local SAFE_DIST = 5

FindLight = Class(BehaviourNode, function(self, inst)
    BehaviourNode._ctor(self, "FindLight")
    self.inst = inst
    self.targ = nil
end)



function FindLight:DBString()
    return string.format("Stay near light %s", tostring(self.targ))
end

function FindLight:Visit()

    if self.status == READY then
        self:PickTarget()
        self.status = RUNNING
    end

    if self.status == RUNNING then

        if self.targ and self.targ:HasTag("fire") then

            local dsq = self.inst:GetDistanceSqToInst(self.targ)

            if dsq >= SAFE_DIST*SAFE_DIST then

 self.inst.components.locomotor:RunInDirection(self.inst:GetAngleToPoint(Point(self.tar
g.Transform:GetWorldPosition())))
            else
                self.inst.components.locomotor:Stop()
                self:Sleep(.5)
            end
        else
            self.status = FAILED
        end
    end
```

```
end

function FindLight:PickTarget()
    self.targ = GetClosestInstWithTag("fire", self.inst, SEE_DIST)
end
```

Follow

```
Follow = Class(BehaviourNode, function(self, inst, target, min_dist, target_dist,
max_dist, canrun)
    BehaviourNode._ctor(self, "Follow")

    self.inst = inst
    self.target = target
    self.min_dist = min_dist
    self.max_dist = max_dist
    self.target_dist = target_dist
    self.canrun = canrun

    if self.canrun == nil then self.canrun = true end

    self.action = "STAND"
end)

function Follow:GetTarget()
    if type(self.target) == "function" then
        return self.target(self.inst)
    end

    return self.target
end


function Follow:DBString()

    local pos = Point(self.inst.Transform:GetWorldPosition())
    local target_pos = Vector3(0,0,0)
    if self.currenttarget then
        target_pos = Point(self.currenttarget.Transform:GetWorldPosition())
    end

    return string.format("%s %s, (%2.2f) ", tostring(self.currenttarget), self.action,
math.sqrt(distsq(target_pos, pos)))
end

function Follow:Visit()

    if self.status == READY then
```

```
        self.currenttarget = self:GetTarget()
        if self.currenttarget then

         local pos = Point(self.inst.Transform:GetWorldPosition())
         local target_pos = Point(self.currenttarget.Transform:GetWorldPosition())
         local dist_sq = distsq(pos, target_pos)

         self.status = RUNNING

         if dist_sq < self.min_dist*self.min_dist then
            self.action = "BACKOFF"
         elseif dist_sq > self.max_dist*self.max_dist then
            self.action = "APPROACH"
         else
            self.status = FAILED
         end

        else
            self.status = FAILED
        end

    end

    if self.status == RUNNING then
        if not self.currenttarget or not self.currenttarget.entity:IsValid()
           or (self.currenttarget.components.health and
self.currenttarget.components.health:IsDead() ) then
            self.status = FAILED
            self.inst.components.locomotor:Stop()
            return
        end


        local pos = Point(self.inst.Transform:GetWorldPosition())
        local target_pos = Point(self.currenttarget.Transform:GetWorldPosition())
        local dist_sq = distsq(pos, target_pos)

        if self.action == "APPROACH" then
            if dist_sq < self.target_dist*self.target_dist then
                self.status = SUCCESS
                return
            end
        elseif self.action == "BACKOFF" then
            if dist_sq > self.target_dist*self.target_dist then
                self.status = SUCCESS
                return
            end
        end
```

```
        if self.action == "APPROACH" then
            local should_run = dist_sq > (self.max_dist*.75)*(self.max_dist*.75)
            local is_running = self.inst.sg:HasStateTag("running")
            if self.canrun and (should_run or is_running) then
                self.inst.components.locomotor:GoToPoint(target_pos, nil, true)
            else
                self.inst.components.locomotor:GoToPoint(target_pos)
            end
        elseif self.action == "BACKOFF" then

         local angle = self.inst:GetAngleToPoint(target_pos)
            if self.canrun then
                self.inst.components.locomotor:RunInDirection(angle + 180)
            else
                self.inst.components.locomotor:WalkInDirection(angle + 180)
            end
        end

        self:Sleep(.25)
    end

end
```

Leash

```
Leash = Class(BehaviourNode, function(self, inst, homelocation, max_dist,
inner_return_dist)
    BehaviourNode._ctor(self, "Leash")
    self.homepos = homelocation
    self.maxdist = max_dist
    self.inst = inst
   self.returndist = inner_return_dist
   self.walking = false
end)


function Leash:Visit()

    if not self:GetHomePos() then
        self.status = FAILED
        return
    end

    if self.status == READY then
      if self:IsInsideLeash() then
         self.status = FAILED
      else
         self.inst.components.locomotor:Stop()
```

```
            self.status = RUNNING
        end
    elseif self.status == RUNNING then
        if self:IsOutsideReturnDist() then
            self.inst.components.locomotor:GoToPoint(self:GetHomePos())
        else
            self.status = SUCCESS
        end
    end
end

function Leash:DBString()
    return string.format("%s, %2.2f", tostring(self:GetHomePos()),
math.sqrt(self:GetDistFromHomeSq() or 0) )
end

function Leash:GetHomePos()
    if type(self.homepos) == "function" then
        return self.homepos(self.inst)
    end

    return self.homepos
end

function Leash:GetDistFromHomeSq()
    local homepos = self:GetHomePos()
   if not homepos then
      return nil
   end
    local pos = Vector3(self.inst.Transform:GetWorldPosition())
    return distsq(homepos, pos)
end

function Leash:IsInsideLeash()
   return self:GetDistFromHomeSq() < self:GetMaxDistSq()
end

function Leash:IsOutsideReturnDist()
   return self:GetDistFromHomeSq() > self:GetReturnDistSq()
end

function Leash:GetMaxDistSq()
    if type(self.maxdist) == "function" then
        local dist = self.maxdist(self.inst)
        return dist*dist
    end

    return self.maxdist*self.maxdist
end
```

```
function Leash:GetReturnDistSq()
    if type(self.returndist) == "function" then
        local dist = self.returndist(self.inst)
        return dist*dist
    end

    return self.returndist*self.returndist
end
```

MinPeriod

```
MinPeriod = Class(BehaviourNode, function(self, inst, minperiod, child)
    BehaviourNode._ctor(self, "MinPeriod", {child})

    self.inst = inst
    self.minperiod = minperiod
end)


function MinPeriod:Visit()
    local child = self.children[1]
   if self.status == READY and self.lastsuccesstime then
      local time = GetTime()
      if time - self.lastsuccesstime < self.minperiod then
         self.status = FAILED
         return
      end
   end

    child:Visit()
    if child.status == SUCCESS then
      self.lastsuccesstime = GetTime()
    end

   self.status = child.status

end

function MinPeriod:DBString()
    if self.minperiod then
      local time = GetTime()


      local time_since_success = time - (self.lastsuccesstime or 0)
      if not self.lastsuccesstime or time_since_success > self.minperiod then
         return string.format("OK (min period is %2.2f)", self.minperiod)
      else
```

```
        return string.format("Waiting for %2.2f (min period is %2.2f)",
self.minperiod-time_since_success, self.minperiod)
      end
    end
end
```

Panic

```
local RUNNING = "running"
local STANDING = "standing"

Panic = Class(BehaviourNode, function(self, inst)
    BehaviourNode._ctor(self, "Panic")
    self.inst = inst
    self.waittime = 0
end)

function Panic:Visit()

    if self.status == READY then
        self:PickNewDirection()
        self.status = RUNNING

    else
        if GetTime() > self.waittime then
            if self.status == RUNNING then
                self:WaitForTime()
            else
                self:PickNewDirection()
            end
        end
        self:Sleep(self.waittime - GetTime())
    end


end

function Panic:WaitForTime()
    self.inst.components.locomotor:Stop()
    self.waittime = GetTime() + 1 + math.random()*2
    self.status = STANDING
end
function Panic:PickNewDirection()
    self.inst.components.locomotor:RunInDirection(math.random()*360)
    self.waittime = GetTime() + 4 + math.random()*2
    self.status = RUNNING
end
```

RunAway

```
RunAway = Class(BehaviourNode, function(self, inst, hunterparams, see_dist, safe_dist,
fn, runhome)
    BehaviourNode._ctor(self, "RunAway")
    self.safe_dist = safe_dist
    self.see_dist = see_dist
    self.hunterparams = hunterparams
    self.inst = inst
    self.runshomewhenchased = runhome
    self.shouldrunfn = fn
end)

function RunAway:__tostring()
    return string.format("RUNAWAY %f from: %s", self.safe_dist, tostring(self.hunter))
end

function RunAway:GetRunAngle(pt, hp)

    if self.avoid_angle then
        local avoid_time = GetTime() - self.avoid_time
        if avoid_time < 1 then
            return self.avoid_angle
        else
            self.avoid_time = nil
            self.avoid_angle = nil
        end
    end

    local angle = self.inst:GetAngleToPoint(hp) + 180 -- + math.random(30)-15
    if angle > 360 then angle = angle - 360 end

    --print(string.format("RunAway:GetRunAngle me: %s, hunter: %s, run: %2.2f",
tostring(pt), tostring(hp), angle))

   local radius = 6

    local result_offset, result_angle, deflected = FindWalkableOffset(pt,
angle*DEGREES, radius, 8, true, false) -- try avoiding walls
    if not result_angle then
        result_offset, result_angle, deflected = FindWalkableOffset(pt, angle*DEGREES,
radius, 8, true, true) -- ok don't try to avoid walls, but at least avoid water
    end
    if not result_angle then
        return angle -- ok whatever, just run
    end

   if result_angle then
      result_angle = result_angle/DEGREES
```

```
      if deflected then
         self.avoid_time = GetTime()
         self.avoid_angle = result_angle
      end
      return result_angle
   end

    return nil
end

function RunAway:Visit()

    if self.status == READY then
      if type(self.hunterparams) == "string" then
         self.hunter = FindEntity(self.inst, self.see_dist, function(guy)
             return not guy:HasTag("notarget")
         end, {self.hunterparams})
      else
         self.hunter = FindEntity(self.inst, self.see_dist, self.hunterparams)
      end

        if self.hunter and self.shouldrunfn and not self.shouldrunfn(self.hunter) then
            self.hunter = nil
        end

        if self.hunter then
            self.status = RUNNING
        else
            self.status = FAILED
        end

    end

    if self.status == RUNNING then
        if not self.hunter or not self.hunter.entity:IsValid() then
            self.status = FAILED
            self.inst.components.locomotor:Stop()
        else

            if self.runshomewhenchased and
              self.inst.components.homeseeker then
               self.inst.components.homeseeker:GoHome(true)
            else
                local pt = Point(self.inst.Transform:GetWorldPosition())
                local hp = Point(self.hunter.Transform:GetWorldPosition())

                local angle = self:GetRunAngle(pt, hp)
                if angle then
                    self.inst.components.locomotor:RunInDirection(angle)
```

```
                else
                    self.status = FAILED
                    self.inst.components.locomotor:Stop()
                end

                if distsq(hp, pt) > self.safe_dist*self.safe_dist then
                    self.status = SUCCESS
                    self.inst.components.locomotor:Stop()
                end
            end


        end
    end
end
```

StandStill

```
StandStill = Class(BehaviourNode, function(self, inst, startfn, keepfn)
    BehaviourNode._ctor(self, "StandStill")
    self.inst = inst
    self.startfn = startfn
    self.keepfn = keepfn
end)


function StandStill:Visit()

    if self.status == READY then
        if not self.startfn or self.startfn(self.inst) then
            self.status = RUNNING
            self.inst.components.locomotor:Stop()
        else
            self.status = FAILED
        end
    end

    if self.status == RUNNING then
        if not self.keepfn or self.keepfn(self.inst) then
            -- yep! standing here is preeeetty great.
            self.inst.components.locomotor:Stop()
        else
            self.status = FAILED
        end
        self:Sleep(.5)
    end

end
```

UseShield

```
UseShield = Class(BehaviourNode, function(self, inst, damageforshield, shieldtime,
hidefromprojectiles)
   BehaviourNode._ctor(self, "UseShield")
   self.inst = inst
   self.damageforshield = damageforshield or 100
   self.hidefromprojectiles = hidefromprojectiles or false
   self.damagetaken = 0
   self.timelastattacked = 1
   self.shieldtime = shieldtime or 2
   self.projectileincoming = false

   self.inst:ListenForEvent("attacked", function(inst, data)
self:OnAttacked(data.attacker, data.damage) end)
   self.inst:ListenForEvent("hostileprojectile", function() self:OnAttacked(nil, 0,
true) end)
   self.inst:ListenForEvent("firedamage", function() self:OnAttacked() end)
   self.inst:ListenForEvent("startfiredamage", function() self:OnAttacked() end)
end)

function UseShield:TimeToEmerge()
   return (GetTime() - self.timelastattacked) > self.shieldtime
end

function UseShield:ShouldShield()
   return (self.damagetaken > self.damageforshield or
self.inst.components.health.takingfiredamage or self.projectileincoming) and not
self.inst.components.health:IsDead()
end

function UseShield:OnAttacked(attacker, damage, projectile)
   if not self.inst.sg:HasStateTag("frozen") then
      self.timelastattacked = GetTime()

      if self.inst.sg.currentstate.name == "shield" and not projectile then
         self.inst.AnimState:PlayAnimation("hit_shield")
         self.inst.AnimState:PushAnimation("hide_loop")
         return
      end

      if damage then
         self.damagetaken = self.damagetaken + damage
      end

      if projectile and self.hidefromprojectiles then
         self.projectileincoming = true
         return
      end
```

```
    end
end

function UseShield:Visit()
    local combat = self.inst.components.combat
    local statename = self.inst.sg.currentstate.name

    if self.status == READY  then
        if self:ShouldShield() or self.inst.sg:HasStateTag("shield") then
            self.damagetaken = 0
            self.projectileincoming = false
            self.inst:PushEvent("entershield")
            --self.inst.sg:GoToState("shield")
            self.status = RUNNING
        else
            self.status = FAILED
        end
    end

    if self.status == RUNNING then
        if not self:TimeToEmerge() or self.inst.components.health.takingfiredamage then
            self.status = RUNNING
        else
            self.inst:PushEvent("exitshield")
            --self.inst.sg:GoToState("shield_end")
            self.status = SUCCESS
        end
    end
end
```

Wander

```
Wander = Class(BehaviourNode, function(self, inst, homelocation, max_dist, times)
    BehaviourNode._ctor(self, "Wander")
    self.homepos = homelocation
    self.maxdist = max_dist
    self.inst = inst
    self.far_from_home = false

    self.times =
    {
      minwalktime = times and times.minwalktime or 2,
      randwalktime = times and times.randwalktime or 3,
      minwaittime = times and times.minwaittime or 1,
      randwaittime = times and times.randwaittime or 3,
    }
end)
```

```
function Wander:Visit()

    if self.status == READY then
        self.inst.components.locomotor:Stop()
      self:Wait(self.times.minwaittime+math.random()*self.times.randwaittime)
        self.walking = false
        self.status = RUNNING
    elseif self.status == RUNNING then

      if not self.walking and self:IsFarFromHome() then
            self:PickNewDirection()
      end

        if GetTime() > self.waittime then
            self:PickNewDirection()
        else
            if not self.walking then
            self:Sleep(self.waittime - GetTime())
            end
        end
    end


end

local function tostring_float(f)
    return f and string.format("%2.2f", f) or tostring(f)
end

function Wander:DBString()
    local w = self.waittime - GetTime()
    return string.format("%s for %2.2f, %s, %s, %s",
        self.walking and 'walk' or 'wait',
        w,
        tostring(self:GetHomePos()),
        tostring_float(math.sqrt(self:GetDistFromHomeSq() or 0)),
        self.far_from_home and "Go Home" or "Go Wherever")
end

function Wander:GetHomePos()
    if type(self.homepos) == "function" then
        return self.homepos(self.inst)
    end

    return self.homepos
end

function Wander:GetDistFromHomeSq()
```

```
    local homepos = self:GetHomePos()
   if not homepos then
      return nil
   end
    local pos = Vector3(self.inst.Transform:GetWorldPosition())
    return distsq(homepos, pos)
end

function Wander:IsFarFromHome()
   if self:GetHomePos() then
      return self:GetDistFromHomeSq() > self:GetMaxDistSq()
   end
   return false
end


function Wander:GetMaxDistSq()
    if type(self.maxdist) == "function" then
        local dist = self.maxdist(self.inst)
        return dist*dist
    end

    return self.maxdist*self.maxdist
end

function Wander:Wait(t)
    self.waittime = t+GetTime()
    self:Sleep(t)
end

function Wander:PickNewDirection()

    self.walking = not self.walking

    self.far_from_home = self:IsFarFromHome()

    if self.walking then

        if self.far_from_home then
            --print(self.inst, Point(self.inst.Transform:GetWorldPosition()), "FAR FROM
HOME", self:GetHomePos())
            self.inst.components.locomotor:GoToPoint(self:GetHomePos())
        else
            local pt = Point(self.inst.Transform:GetWorldPosition())
            local angle = math.random()*2*PI
            local radius = 12
            local attempts = 8
            local offset, check_angle, deflected = FindWalkableOffset(pt, angle,
radius, attempts, true, false) -- try to avoid walls
```

```
            if not check_angle then
                --print(self.inst, "no los wander, fallback to ignoring walls")
                offset, check_angle, deflected = FindWalkableOffset(pt, angle, radius,
attempts, true, true) -- if we can't avoid walls, at least avoid water
            end
            if check_angle then
                angle = check_angle
            else
                -- guess we don't have a better direction, just go whereever
                --print(self.inst, "no walkdable wander, fall back to random")
            end
            --print(self.inst, pt, string.format("wander to %s @ %2.2f %s",
tostring(offset), angle/DEGREES, deflected and "(deflected)" or ""))
            self.inst.components.locomotor:WalkInDirection(angle/DEGREES)
        end

        self:Wait(self.times.minwalktime+math.random()*self.times.randwalktime)
    else
        self.inst.components.locomotor:Stop()

        --if self.far_from_home then
            --self:Wait(1+math.random())
        --else
            self:Wait(self.times.minwaittime+math.random()*self.times.randwaittime)
        --end
    end

end
```