Appendix I: game.py (Oversees all object interactions.)

```
import pygame
from pygame.locals import *
from beaver import Beaver
from brain import Brain
from constants import Constants
from parameters import GameParameters
from marsh import Marsh
from terrain import Terrain
from tree import Tree
from wolf import Wolf

BG_COLOR = GameParameters.BG_COLOR
HEALTHBAR_COLOR = GameParameters.HEALTHBAR_COLOR
HEALTHBAR_HEIGHT = GameParameters.HEALTHBAR_HEIGHT

SCREEN_WIDTH = GameParameters.SCREEN_WIDTH
SCREEN_HEIGHT = GameParameters.SCREEN_HEIGHT

FRAMERATE = GameParameters.FRAMERATE

NUM_GENERATIONS = GameParameters.NUM_GENERATIONS

class Game:
  def __init__(self):
    self._running = True
    self.screen = None
    self.size = self.width, self.height = SCREEN_WIDTH, SCREEN_HEIGHT

  def on_init(self):
    pygame.init()
    self.screen = pygame.display.set_mode(
      self.size, pygame.HWSURFACE | pygame.DOUBLEBUF)

    # Fill background and blits everything to the screen
    self.background = pygame.Surface(self.size)
    self.background = self.background.convert()
    self.background.fill(BG_COLOR)
    self.screen.blit(self.background, (0, 0))
    pygame.display.flip()

    self.terrain = Terrain()

    self.beaver = Beaver()
    self.beaversprite = pygame.sprite.RenderPlain(self.beaver)
    self.generationtime = pygame.time.get_ticks()

    self.brain = Brain()
    self.brain.environment.setbeaver(self.beaver)

    self.wolf = Wolf()
    self.wolfsprite = pygame.sprite.RenderPlain(self.wolf)

    self._clock = pygame.time.Clock()
```

```
    self._running = True

  def on_event(self, event):
    if event.type == pygame.QUIT:
      self._running = False

  def on_loop(self):
    self.beaver.seteyeview(self.terrain.terraingroup)
    self.beaversprite.update()

    self.brain.experiment.doInteractions(1)

    self.wolf.seteyeview(self.terrain.terraingroup)
    self.wolf.setscentview(self.beaver)
    #self.wolfsprite.update()

    marsh = self.terrain.getmarsh()
    if (self.beaver.action == Constants.BEAVER_ACTION_DROP_LUMBER and
        self.beaver.droppedlumber and
        pygame.sprite.collide_rect(self.beaver, marsh)):
      marsh.improve()
    marsh.update()

    # Reset the wolf if it gets stuck in marsh
    if pygame.sprite.collide_rect(self.wolf, marsh):
      self.wolf.respawn()

    if (self.beaver.energy <= 0 or
      self.beaver.rect.colliderect(self.wolf.rect)):

      temp = pygame.time.get_ticks()
      # Only when beaver starves
      if self.beaver.energy <= 0:
        generationtimes.append("%d\t%d" % (self.beaver.generationcount,
                                           temp - self.generationtime))
      self.generationtime = temp

      self.beaver.respawn()
      self.brain.agent.learn()
      self.brain.agent.reset()




      if self.beaver.generationcount > NUM_GENERATIONS:
        self._running = False

      # Reset the wolf so that it seems as if time has passed
      # (aka wolf not lurking around marsh on beaver spawn)
      self.wolf.respawn()

      # Reset the environment so beavers start alike.
      marsh.respawn()
      self.terrain.respawntrees()
```

```
    else:
      tree = pygame.sprite.spritecollideany(self.beaver,
        self.terrain.gettreelist())
      if tree is not None and not isinstance(tree, Marsh):
        # Check beaver state
        if self.beaver.action == Constants.BEAVER_ACTION_EAT:
          tree.setstate(Constants.TREE_STATE_ATE)
          tree.update()
        elif (self.beaver.action == Constants.BEAVER_ACTION_PICK_UP_LUMBER and
              self.beaver.pickeduplumber):
          tree.setstate(Constants.TREE_STATE_FORAGED)
          tree.update()

        # Check tree state
        if tree.health <= 0:
          self.terrain.respawntree(tree)

  def on_render(self):
    self.background.fill(BG_COLOR)
    self.screen.blit(self.background, (0, 0))

    # Draws beaver, wolf, marsh, and tree sprites
    self.terrain.terraingroup.draw(self.screen)
    self.beaversprite.draw(self.screen)
    self.wolfsprite.draw(self.screen)

    # Draws energy and health bars of beaver and trees
    bx, by = self.beaver.rect.topleft
    brect = pygame.Rect(bx, by, self.beaver.energybar, HEALTHBAR_HEIGHT)
    pygame.draw.rect(self.screen, HEALTHBAR_COLOR, brect, 0)

    for sprite in self.terrain.terraingroup:
      sx, sy = sprite.rect.topleft
      srect = pygame.Rect(sx, sy, sprite.healthbar, HEALTHBAR_HEIGHT)
      pygame.draw.rect(self.screen, HEALTHBAR_COLOR, srect, 0)

    # Inefficient but works w/o hacking up a blit function for transparent imgs
    pygame.display.update()
    self._clock.tick(FRAMERATE)

  def on_cleanup(self):
    pygame.quit()

  def on_execute(self):
    if self.on_init() == False:
      self._running = False

    while (self._running):
      for event in pygame.event.get():
        self.on_event(event)
      self.on_loop()
      self.on_render()
    self.on_cleanup()
```

```
if __name__ == "__main__":
  generationtimes = []
  game = Game()
  game.on_execute()
  with open('generationtime.txt', 'a') as datafile:
    datafile.write(','.join(generationtimes))
    datafile.write('\n')
```

Appendix II: beaver.py (Beaver class that defines its attributes, states, and actions.)

```
import math
import pygame
import random
from pygame.locals import *
from operator import itemgetter
from constants import Constants
from marsh import Marsh
from resources import Resources
from tree import Tree
from parameters import BeaverParameters

CONST_VIEW_DIST = BeaverParameters.CONST_VIEW_DIST
CONST_SCENT_DIST = BeaverParameters.CONST_SCENT_DIST
CONST_STEP_SIZE_LAND = BeaverParameters.CONST_STEP_SIZE_LAND
CONST_STEP_SIZE_WATER = BeaverParameters.CONST_STEP_SIZE_WATER

CONST_LUMBER_WEIGHT = BeaverParameters.CONST_LUMBER_WEIGHT

CONST_INITIAL_ENERGY = BeaverParameters.CONST_INITIAL_ENERGY
CONST_MAX_ENERGY = BeaverParameters.CONST_MAX_ENERGY

CONST_DEAD_ENERGY_THRESHOLD = BeaverParameters.CONST_DEAD_ENERGY_THRESHOLD
CONST_LOW_ENERGY_THRESHOLD = BeaverParameters.CONST_LOW_ENERGY_THRESHOLD
CONST_MED_ENERGY_THRESHOLD = BeaverParameters.CONST_MED_ENERGY_THRESHOLD

CONST_ENERGY_IDLE_COST = BeaverParameters.CONST_ENERGY_IDLE_COST
CONST_ENERGY_WALK_LAND_COST = BeaverParameters.CONST_ENERGY_WALK_LAND_COST
CONST_ENERGY_WALK_WATER_COST = BeaverParameters.CONST_ENERGY_WALK_WATER_COST
CONST_ENERGY_EAT_GAIN = BeaverParameters.CONST_ENERGY_EAT_GAIN
CONST_ENERGY_PICK_UP_LUMBER_COST = BeaverParameters.CONST_ENERGY_PICK_UP_LUMBER_COST

class Beaver(pygame.sprite.Sprite):
  """A beaver that will move across the screen
  Returns: beaver object
  Functions: update, calcnewpos
  Attributes: action, adjlist, energy, energybar, eyeview, haslumber, inwater, rect,
    scentview, states, stepsize
  """


  # No cost in dropping lumber

  def __init__(self):
    self.generationcount = 1
```

```
    self.reset()

  def reset(self):
    # Centers the beaver to spawn in the center of the marshes
    pygame.sprite.Sprite.__init__(self)
    self.image, self.rect = Resources.load_png('beaver.png')
    originalsize = self.image.get_size()
    self.image = pygame.transform.scale(
      self.image, (int(originalsize[0]/2), int(originalsize[1]/2)))
    self.rect = self.image.get_rect()
    newsize = self.image.get_size()
    screen = pygame.display.get_surface()
    centerx = screen.get_width()/2 - newsize[0]/2
    centery = screen.get_height()/2 - newsize[1]/2
    self.rect.move_ip(centerx, centery)

    self.action = Constants.BEAVER_ACTION_MOVE_TREE
    self.energy = CONST_INITIAL_ENERGY
    self.energybar = self.rect.width
    self.eyeview = [] # Contains knowledge of nearby sprites by vision
    self.haslumber = False
    self.pickeduplumber = False
    self.droppedlumber = False
    self.inwater = True # Beaver spawns in marsh
    self.scentview = [] # Contains knowledge of nearby wolf by scent
    self.states = [Constants.BEAVER_STATE_BEAVER_ENERGY_HIGH,
      Constants.BEAVER_STATE_MARSH_HEALTH_LOW,
      Constants.BEAVER_STATE_NO_LUMBER,
      Constants.BEAVER_STATE_NONE_TREE,
      Constants.BEAVER_STATE_AT_MARSH,
      Constants.BEAVER_STATE_NONE_WOLF]
    self.stepsize = CONST_STEP_SIZE_WATER

    # Top left, top, top right, left, right, bottom left, bottom, bottom right
    self.setadjpoints()

  def respawn(self):
    self.generationcount += 1
    self.reset()

  def setadjpoints(self):
    self.adjpoints = [
      (self.rect.centerx - self.stepsize, # top left
      self.rect.centery - self.stepsize),
      (self.rect.centerx, # top
      self.rect.centery - self.stepsize),
      (self.rect.centerx + self.stepsize, # top right
      self.rect.centery - self.stepsize),
      (self.rect.centerx - self.stepsize, # left
      self.rect.centery),
      (self.rect.centerx + self.stepsize, # right
      self.rect.centery),
      (self.rect.centerx - self.stepsize, # bottom left
      self.rect.centery + self.stepsize),
```

```
      (self.rect.centerx, # bottom
      self.rect.centery + self.stepsize),
      (self.rect.centerx + self.stepsize, # bottom right
      self.rect.centery + self.stepsize)]

  def setbrain(self, brain):
    self.brain = brain

  def setaction(self, action):
    #print "beaver action is changed to " + str(action)
    self.action = action

  def setstate(self, index, state):
    self.states[index] = state

  def setstepsize(self, stepsize):
    self.stepsize = stepsize

  """The beaver can observe trees within a 100x100 rect.
  Saves trees within eye viewing distance into internal list.
  """
  def seteyeview(self, terrain):
    x = self.rect.centerx - CONST_VIEW_DIST
    y = self.rect.centery - CONST_VIEW_DIST
    eyeviewrect = Rect(x, y, CONST_VIEW_DIST*2, CONST_VIEW_DIST*2)
    self.eyeview = []
    for sprite in terrain:
      if eyeviewrect.colliderect(sprite.rect):
        self.eyeview.append(sprite)

  def setscentview(self, wolf):
    x = self.rect.centerx - CONST_SCENT_DIST
    y = self.rect.centery - CONST_SCENT_DIST
    scentviewrect = Rect(x, y, CONST_SCENT_DIST*2,
      CONST_SCENT_DIST*2)
    self.scentview = []
    if scentviewrect.colliderect(wolf.rect):
      self.scentview.append(wolf)
      self.setstate(Constants.BEAVER_STATE_INDEX_WOLF,
        Constants.BEAVER_STATE_SEE_WOLF)
    else:
      self.setstate(Constants.BEAVER_STATE_INDEX_WOLF,
        Constants.BEAVER_STATE_NONE_WOLF)

  def gettreedisttuple(self, spritelist, point):
    treedisttuple = []
    for sprite in spritelist:
      if isinstance(sprite, Tree):
        treedisttuple.append((sprite,
          Resources.calcdistance(point, sprite.rect.center)))
    return treedisttuple

  def gettreeview(self, view):
    treeinfo = []
```

```
    for sprite in view:
      if isinstance(sprite, Tree):
        treeinfo.append(sprite)
    return treeinfo

  """Return a list of eight values for the beaver's adjacency list.
  High values are favored. In the future, we may either influence these
  values based on distance from home or learn it.
  """
  def calcadjvalsfood(self):
    adjvals = []
    treeinfo = self.gettreeview(self.eyeview)
    if treeinfo:
      for point in self.adjpoints:
        treedisttuple = self.gettreedisttuple(treeinfo, point)
        # Get the distance to the closest tree
        # Sorting may be useful for later ops
        shortestdist = sorted(treedisttuple, key=itemgetter(1))[0][1]
        normalizeddist = shortestdist/(CONST_VIEW_DIST * math.sqrt(2) +
          math.sqrt(pow(treeinfo[0].rect.width, 2) + pow(treeinfo[0].rect.height, 2)))
        adjvals.append(1 - normalizeddist)
    return adjvals

  def calcadjvalspred(self):
    adjvals = []
    if self.scentview:
      scentpoint = self.scentview[0].rect.center
      for point in self.adjpoints:
        scentdisttuple = [(self.scentview[0],
          Resources.calcdistance(scentpoint, point))]
        shortestdist = sorted(scentdisttuple, key=itemgetter(1))[0][1]
        normalizeddist = shortestdist/(CONST_VIEW_DIST * math.sqrt(2))
        adjvals.append(1 - normalizeddist)
    return adjvals

  def calcadjvalsmarsh(self):
    adjvals = []
    for sprite in self.eyeview:
      if isinstance(sprite, Marsh):
        marshpoint = sprite.rect.center
        for point in self.adjpoints:
          marshdisttuple = [(sprite,
            Resources.calcdistance(marshpoint, point))]
          shortestdist = sorted(marshdisttuple, key=itemgetter(1))[0][1]
          # TODO: May have to tweak this normalized distance
          normalizeddist = shortestdist/(CONST_VIEW_DIST * math.sqrt(2))
          adjvals.append(1 - normalizeddist)
    return adjvals

  # Obsolete method used for NN
  def calcnewpos(self, rect):
    if self.action == Constants.BEAVER_ACTION_MOVE_TREE:
      self.setadjpoints()
      adjvalsfood = self.calcadjvalsfood()
```

```
    adjvalspred = self.calcadjvalspred()
    adjvalsmarsh = self.calcadjvalsmarsh()
    #adjpointidx = self.brain.getmaxadjidx(adjvalsfood, adjvalspred, adjvalsmarsh)

    if adjvalsfood:
      #print "Using brain max in homing " + str(adjpointidx)
      moveto = self.adjpoints[adjvalsfood.index(max(adjvalsfood))]
      #moveto = self.adjpoints[adjpointidx]
      print "Fixed moving from " + str(self.rect.center) + " moving to " + str(moveto) + '\n'
      # Note that the move function returns a new rect moved by offset
      offsetx = moveto[0] - self.rect.width/2 - self.rect.x
      offsety = moveto[1] - self.rect.height/2 - self.rect.y
      return rect.move(offsetx, offsety)
    else: # Pick random location to move to if can't view any nearby trees
      #print "Moving randomly and not using " + str(adjpointidx) + '\n'
      while True:
        offsetx = (random.randint(0, 1)*2 - 1) * self.stepsize
        offsety = (random.randint(0, 1)*2 - 1) * self.stepsize
        newx = self.rect.x + offsetx - self.rect.width/2
        newy = self.rect.y + offsety - self.rect.height/2
        if newx >= 0 and newy >= 0:
          break
      return rect.move(offsetx, offsety)
  else: # CONST_STATE_EAT or CONST_STATE_FORAGE
    return self.rect # Don't move

# Use this method only if knows there is a tree nearby and not at tree already
# If violation of rules when method called, beaver doesn't move and still lose energy.
def performactionmovetotree(self):
  self.setaction(Constants.BEAVER_ACTION_MOVE_TREE)
  self.setadjpoints()
  adjvalsfood = self.calcadjvalsfood()
  if (self.gettreeview(self.eyeview) and
    self.rect.collidelist(self.gettreeview(self.eyeview)) >= 0):
    print "performactionmovetotree: already at tree!"
    self.energy -= CONST_ENERGY_IDLE_COST
  elif adjvalsfood:
    moveto = self.adjpoints[adjvalsfood.index(max(adjvalsfood))]
    offsetx = moveto[0] - self.rect.width/2 - self.rect.x
    offsety = moveto[1] - self.rect.height/2 - self.rect.y
    if self.haslumber:
      if self.inwater:
        self.energy -= (CONST_ENERGY_WALK_WATER_COST *
          CONST_LUMBER_WEIGHT)
      else:
        self.energy -= (CONST_ENERGY_WALK_LAND_COST *
          CONST_LUMBER_WEIGHT)
    else:
      if self.inwater:
        self.energy -= CONST_ENERGY_WALK_WATER_COST
      else:
        self.energy -= CONST_ENERGY_WALK_LAND_COST
    self.rect = self.rect.move(offsetx, offsety)
  else:
```

```
      print "performactionmovetotree: no trees in sight - unable to move to nearest tree"
      self.energy -= CONST_ENERGY_IDLE_COST

  # Use this method only if knows there is a marsh nearby and not in marsh already
  # If violation of rules when method called, beaver doesn't move and still loses energy.
  def performactionmovetomarsh(self):
    self.setaction(Constants.BEAVER_ACTION_MOVE_MARSH)
    self.setadjpoints()
    adjvalsmarsh = self.calcadjvalsmarsh()
    if self.inwater:
      print "performactionmovetomarsh: already on marsh!"
      self.energy -= CONST_ENERGY_IDLE_COST
    elif adjvalsmarsh:
      moveto = self.adjpoints[adjvalsmarsh.index(max(adjvalsmarsh))]
      offsetx = moveto[0] - self.rect.width/2 - self.rect.x
      offsety = moveto[1] - self.rect.height/2 - self.rect.y
      if self.haslumber:
        self.energy -= (CONST_ENERGY_WALK_LAND_COST *
          CONST_LUMBER_WEIGHT)
      else:
        self.energy -= CONST_ENERGY_WALK_LAND_COST # Moving on land
      self.rect = self.rect.move(offsetx, offsety)
    else:
      print "performactionmovetomarsh: no marsh in sight - unable to move to marsh"
      self.energy -= CONST_ENERGY_IDLE_COST

  # Use this method only if beaver is at a tree
  def performactioneat(self):
    self.setaction(Constants.BEAVER_ACTION_EAT)
    if (self.gettreeview(self.eyeview) and
      self.rect.collidelist(self.gettreeview(self.eyeview)) >= 0):
      self.energy += CONST_ENERGY_EAT_GAIN

      if self.energy > CONST_MAX_ENERGY:
        self.energy = CONST_MAX_ENERGY

    else:
      print "performactioneat: not at tree - cannot eat"
      self.energy -= CONST_ENERGY_IDLE_COST

  def performactionpickuplumber(self):
    self.setaction(Constants.BEAVER_ACTION_PICK_UP_LUMBER)
    self.pickeduplumber = False
    if (not self.haslumber and
        self.gettreeview(self.eyeview) and
        self.rect.collidelist(self.gettreeview(self.eyeview)) >= 0):
      self.haslumber = True
      self.pickeduplumber = True
      self.setstate(Constants.BEAVER_STATE_INDEX_LUMBER,
        Constants.BEAVER_STATE_HAS_LUMBER)
      self.energy -= CONST_ENERGY_PICK_UP_LUMBER_COST
    elif self.haslumber:
      print "performactionpickuplumber: has lumber - cannot pick up lumber"
      self.energy -= CONST_ENERGY_IDLE_COST
```

```
    else:
      print "performactionpickuplumber: not at tree - cannot pick up lumber"
      self.energy -= CONST_ENERGY_IDLE_COST

  # Can drop lumber anywhere resulting in 0 energy change; doesn't have to be in marsh to drop it
  def performactiondroplumber(self):
    self.droppedlumber = False
    if self.haslumber:
      self.setaction(Constants.BEAVER_ACTION_DROP_LUMBER)
      self.haslumber = False
      self.droppedlumber = True
      self.setstate(Constants.BEAVER_STATE_INDEX_LUMBER,
        Constants.BEAVER_STATE_NO_LUMBER)
    else:
      print "performactiondroplumber: does not have lumber - cannot drop lumber"
      self.energy -= CONST_ENERGY_IDLE_COST

  def performaction(self, action):
    action = int(action)
    print "performing an action"
    if action == Constants.BEAVER_ACTION_INDEX_MOVE_TREE:
      self.performactionmovetotree()
    elif action == Constants.BEAVER_ACTION_INDEX_MOVE_MARSH:
      self.performactionmovetomarsh()
    elif action == Constants.BEAVER_ACTION_INDEX_EAT:
      self.performactioneat()
    elif action == Constants.BEAVER_ACTION_INDEX_PICK_UP_LUMBER:
      self.performactionpickuplumber()
    elif action == Constants.BEAVER_ACTION_INDEX_DROP_LUMBER:
      self.performactiondroplumber()
    else:
      print "Invalid action index: " + str(action)

    self.updateenergy()
    print self.energy

  def updateenergy(self):
    if self.energy == CONST_DEAD_ENERGY_THRESHOLD:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_ZERO)
    elif self.energy < CONST_LOW_ENERGY_THRESHOLD:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_LOW)
    elif self.energy < CONST_MED_ENERGY_THRESHOLD:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_MED)
    else:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_HIGH)

    self.energybar = self.rect.width * min(1, (self.energy/100.0))

  def updateenergynobrain(self):
    if self.action == Constants.BEAVER_ACTION_MOVE_TREE:
```

```
      if self.inwater:
        self.energy -= CONST_ENERGY_WALK_WATER_COST
      else:
        self.energy -= CONST_ENERGY_WALK_LAND_COST
    elif self.action == Constants.BEAVER_ACTION_EAT:
      self.energy += CONST_ENERGY_EAT_GAIN
    elif self.action == Constants.BEAVER_ACTION_PICK_UP_LUMBER:
      self.energy -= CONST_ENERGY_PICK_UP_LUMBER_COST

    # Set beaver energy state
    if self.energy == CONST_DEAD_ENERGY_THRESHOLD:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_ZERO)
    elif self.energy < CONST_LOW_ENERGY_THRESHOLD:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_LOW)
    elif self.energy < CONST_MED_ENERGY_THRESHOLD:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_MED)
    else:
      self.setstate(Constants.BEAVER_STATE_INDEX_BEAVER_ENERGY,
        Constants.BEAVER_STATE_BEAVER_ENERGY_HIGH)

    self.energybar = self.rect.width * min(1, (self.energy/100.0))

  def updatetreeresponse(self):
    self.setstate(Constants.BEAVER_STATE_INDEX_TREE,
      Constants.BEAVER_STATE_NONE_TREE)
    if self.gettreeview(self.eyeview):
      self.setstate(Constants.BEAVER_STATE_INDEX_TREE,
        Constants.BEAVER_STATE_SEE_TREE)
      if self.rect.collidelist(self.gettreeview(self.eyeview)) >= 0:
        self.setstate(Constants.BEAVER_STATE_INDEX_TREE,
          Constants.BEAVER_STATE_AT_TREE)

  def updateterraintyperesponse(self):
    # If beaver does not see marsh, assume it retains previous knowledge of it
    self.inwater = False
    self.setstepsize(CONST_STEP_SIZE_LAND)
    self.setstate(Constants.BEAVER_STATE_INDEX_MARSH,
      Constants.BEAVER_STATE_NONE_MARSH)
    for sprite in self.eyeview:
      if isinstance(sprite, Marsh):
        self.setstate(Constants.BEAVER_STATE_INDEX_MARSH_HEALTH,
          sprite.gethealthlevel())
        self.setstate(Constants.BEAVER_STATE_INDEX_MARSH,
          Constants.BEAVER_STATE_SEE_MARSH)
        if pygame.sprite.collide_rect(self, sprite):
          self.inwater = True
          self.setstepsize(CONST_STEP_SIZE_WATER)
          self.setstate(Constants.BEAVER_STATE_INDEX_MARSH,
            Constants.BEAVER_STATE_AT_MARSH)

  def update(self):
```

```
    # First, update tree response
    self.updatetreeresponse()

    # Second, update terrain type response
    self.updateterraintyperesponse()

    # No brain movement
    """if self.rect.collidelist(self.gettreeview(self.eyeview)) >= 0:
      self.setaction(Constants.BEAVER_ACTION_EAT)
    newpos = self.calcnewpos(self.rect)
    self.rect = newpos
    self.updateenergynobrain()"""
```

Appendix III: derived_constants.py (Creates the dictionary that indexes all the possible combinations of states.)

```
from constants import Constants
from parameters import StateWeightParameters

def get_num_states():
  return (len(StateWeightParameters.BEAVER_ENERGY.keys()) *
          len(StateWeightParameters.MARSH_ENERGY.keys()) *
          len(StateWeightParameters.LUMBER.keys()) *
          len(StateWeightParameters.TREE.keys()) *
          len(StateWeightParameters.MARSH.keys()) *
          len(StateWeightParameters.WOLF.keys()))

def get_num_actions():
  return len(StateWeightParameters.ACTION.keys())

def get_state_to_index_and_rewards():
  only_states = []
  rewards = {}
  for beaver_state in StateWeightParameters.BEAVER_ENERGY.iteritems():
    for marsh_state in StateWeightParameters.MARSH_ENERGY.iteritems():
      for lumber_state in StateWeightParameters.LUMBER.iteritems():
        for env_tree_state in StateWeightParameters.TREE.iteritems():
          for env_marsh_state in StateWeightParameters.MARSH.iteritems():
            for env_wolf_state in StateWeightParameters.WOLF.iteritems():
              only_state = (
                beaver_state[0], marsh_state[0], lumber_state[0],
                env_tree_state[0], env_marsh_state[0], env_wolf_state[0])
              only_states.append(only_state)

              for action in StateWeightParameters.ACTION.iteritems():
                full_state = (
                  beaver_state[0], marsh_state[0], lumber_state[0],
                  env_tree_state[0], env_marsh_state[0], env_wolf_state[0],
                  action[0])

                if ((action[0] == Constants.BEAVER_ACTION_EAT and env_tree_state[0] != Constants.BEAVER_STATE_AT_TREE) or
                   (action[0] == Constants.BEAVER_ACTION_PICK_UP_LUMBER and env_tree_state[0] != Constants.
                   BEAVER_STATE_AT_TREE) or
                   (action[0] == Constants.BEAVER_ACTION_PICK_UP_LUMBER and lumber_state[0] != Constants.
                   BEAVER_STATE_NO_LUMBER) or
```

```
                    (action[0] == Constants.BEAVER_ACTION_DROP_LUMBER and lumber_state[0] != Constants.
                    BEAVER_STATE_HAS_LUMBER) or
                    (action[0] == Constants.BEAVER_ACTION_MOVE_TREE and env_tree_state[0] != Constants.
                    BEAVER_STATE_SEE_TREE) or
                    (action[0] == Constants.BEAVER_ACTION_MOVE_MARSH and env_marsh_state[0] != Constants.
                    BEAVER_STATE_SEE_MARSH)):
                    rewards[full_state] = 0
                else:
                  rewards[full_state] = (
                    beaver_state[1] * marsh_state[1] * lumber_state[1] *
                    env_tree_state[1] * env_marsh_state[1] * env_wolf_state[1] *
                    action[1])

  index_to_state = sorted(only_states)
  state_to_index = {state: index for index, state in enumerate(index_to_state)}
  return (state_to_index, rewards)

class DerivedConstants:
  NUM_STATES = get_num_states()
  NUM_ACTIONS = get_num_actions()
  STATE_TO_INDEX, REWARDS = get_state_to_index_and_rewards()
```