Game AI

State Machines
AI

- Complex AI != Good AI
- ex) pacman - ghost at each corner: randomly choose between moving toward player, or moving in random direction.
- predictable == boring
- too good == no fun
- keep imperfections, but be challenging for users
Types of AIs

- Agent
  - virtual character (enemy, NPC, sidekick, cow)
  - structure
    - sensor/input - gather info on surroundings (enemies, map)
    - working memory (STM) - state (location, orientation, FSM state), paths, etc.
    - reasoning/analysis - use state + sensors to decide (FSMs, rule systems, etc)
    - action/output - steering, shooting, etc (make characters intentions/emotions obvious, look intelligent)

- Abstract Controller
  - group dynamics
  - higher level than character
Finite State Machines (FSMs)

- FSM == deterministic finite state automata
- FSM = states + transitions
- easy to understand, efficient, widely applicable
- start with a diagram before coding!
- identify behaviors
Patrolling Soldier FSM

- outdoors, no obstacles
- patrols 'waypoints', in order
- activates if enemy visible
- if sees enemy, chases
- if close, engage in combat
Parallel Automata

- for complex behavior
- model as concurrent, simple behaviors (separate brains)
- ex) moving and shooting
Synchronized FSMs

• for inter-automata communication
  • ex) small teams
• bulletin board pattern
  • shared memory accessed by FSMs
  • use to post and read messages about individual state/events
• ex) first soldier to see enemy is attacker:
  • waits for cover from 2nd soldier, then attacks.
  • 3rd soldier hands back and lobbs grenades
  • role depends on who sees enemy first
Team Tactics
Synchronized FSM

- bulletin board state (Booleans):
  - fighting
  - attacker_alive
  - cover_alive
  - grenade_alive
  - request_cover
  - cover
Non-deterministic FSMs

- choose transition randomly, given probabilities
- not as predictable

\[ \text{Smoke} \quad \rightarrow \quad \text{Turn} \]

\[ \text{Seek waypoint} \quad \rightarrow \quad \text{Chase} \]

\[ \text{dead} \quad (1.0) \quad \rightarrow \quad \text{not aligned} \quad (1.0) \]

\[ \text{see player} \quad (0.5) \quad \rightarrow \quad \text{aligned and distance < 50} \quad (1.0) \]

\[ \text{reached} \quad (0.5) \quad \rightarrow \quad \text{done} \quad (1.0) \]

\[ \text{done} \quad (1.0) \quad \rightarrow \quad \text{reached} \quad (0.5) \]

\[ \text{see player} \quad (0.5) \quad \rightarrow \quad \text{not aligned} \quad (1.0) \]

\[ \text{dead} \quad (1.0) \quad \rightarrow \quad \text{not aligned} \quad (1.0) \]

\[ \text{Monday, May 11, 2009} \]
We also need to define the links that connect states together. These take the form of a condition and the name of the state to switch to if the condition is met. The exploring state, for example, has two such links (see Table 7-2).

Table 7-2. Links from Exploring State

<table>
<thead>
<tr>
<th>Condition</th>
<th>Destination State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seen a leaf</td>
<td>Seeking</td>
</tr>
<tr>
<td>Spider attacking base!</td>
<td>Hunting</td>
</tr>
<tr>
<td>Leaf gone</td>
<td>Exploring</td>
</tr>
<tr>
<td>Got a leaf</td>
<td>Hunting</td>
</tr>
<tr>
<td>In nest</td>
<td>Delivering</td>
</tr>
<tr>
<td>Spider gone</td>
<td>Delivering</td>
</tr>
<tr>
<td>Spider dead</td>
<td>Delivering</td>
</tr>
</tbody>
</table>

Once we have defined the links between the states, we have a state machine that can be used as the brain for an entity. Figure 7-3 shows the complete state machine that we will be building for the ant. Drawing a state machine out on paper like this is a great way of visualizing how it all fits together, and will help you when you need to turn it into code.

Let's put this into practice and create the code for the state machine. We will begin by defining a base class for an individual state (Listing 7-6). Later we will create another class for the state machine as a whole that will manage the states it contains.

The base `State` class doesn't actually do anything other than store the name of the state in the constructor. The remaining functions in `State` do nothing—the `pass` keyword simply tells Python that you intentionally left the function blank. We need these empty functions because not all of the states we will be building will implement all of the functions in the base class. The exploring state, for example, has no exit actions. When we come to implement the `AntStateExploring` class, we can omit the `exit_actions` function because it will safely fall back to the do-nothing version of the function in the base class (`State`).
Implementing the State Pattern

class State(object):
    def __init__(self, name):
        self.name = name
    def do_actions(self):
        pass
    def check_conditions(self):
        pass
    def entry_actions(self):
        pass
    def exit_actions(self):
        pass

class StateMachine:
    states
    active_state
    def add_state(self, state):
        pass
    def think(self):
        pass
    def set_state(self):
        pass
    def do_actions(self):
        pass
    def check_conditions(self):
        pass
    def entry_actions(self):
        pass
    def exit_actions(self):
        pass
class StateMachine(object):
    
    def __init__(self):
        self.states = {}
        self.active_state = None

    def add_state(self, state):
        self.states[state.name] = state

    def think(self):
        if self.active_state is None:
            return
        self.active_state.do_actions()
        new_state_name = self.active_state.check_conditions()
        if new_state_name is not None:
            self.set_state(new_state_name)

    def set_state(self, new_state_name):
        if self.active_state is not None:
            self.active_state.exit_actions()
        self.active_state = self.states[new_state_name]
        self.active_state.entry_actions()
State

StateMachine

GameEntity

world
name
image
location
destination
speed
brain : StateMachine
id

+ render(surface)
+ process(time_passed)

World

entities
entity_id
background

+ add_entity(entity)
+ remove_entity(entity)
+ get(entity_id)
+ process(time_passed)
+ render(surface)
+ get_close_entity(name, location, range=100.)

GameEntity

Ant

@seeking_state
@delivering_state
@exploring_state
@hunting_state

+ carry(image)
+ drop(surface)
+ render(surface)
+ process(time_passed)

Spider

health
dead_image

+ bitten()
+ render(surface)
+ process(time_passed)

StateMachine

Leaf

AntStateExploring

AntStateDelivering

AntStateHunting

AntStateSeeking

State

Monday, May 11, 2009
class GameEntity(object):

    def __init__(self, world, name, image):
        self.world = world
        self.name = name
        self.image = image
        self.location = Vector2(0, 0)
        self.destination = Vector2(0, 0)
        self.speed = 0.

        self.brain = StateMachine()
        self.id = 0

    def render(self, surface):
        x, y = self.location
        w, h = self.image.get_size()
        surface.blit(self.image, (x-w/2, y-h/2))

    def process(self, time_passed):

        self.brain.think()

        if self.speed > 0. and self.location != self.destination:

            vec_to_destination = self.destination - self.location
            distance_to_destination = vec_to_destination.get_length()
            heading = vec_to_destination.get_normalized()
            travel_distance = min(distance_to_destination, time_passed * self.speed)
            self.location += travel_distance * heading
class StateMachine(object):

    def __init__(self):
        self.states = {}
        self.active_state = None

    def add_state(self, state):
        self.states[state.name] = state

    def think(self):
        if self.active_state is None:
            return

        self.active_state.do_actions()

        new_state_name = self.active_state.check_conditions()
        if new_state_name is not None:
            self.set_state(new_state_name)

    def set_state(self, new_state_name):
        if self.active_state is not None:
            self.active_state.exit_actions()

        self.active_state = self.states[new_state_name]
        self.active_state.entry_actions()
```python
class World(object):
    def __init__(self):
        self.entities = []
        self.entity_id = 0
        self.background = pygame.surface.Surface(SCREEN_SIZE).convert()
        self.background.fill((255, 255, 255))
        pygame.draw.circle(self.background, (200, 255, 200), NEST_POSITION, int(NEST_SIZE))

    def add_entity(self, entity):
        self.entities[self.entity_id] = entity
        entity.id = self.entity_id
        self.entity_id += 1

    def remove_entity(self, entity):
        del self.entities[entity.id]

    def get(self, entity_id):
        if entity_id in self.entities:
            return self.entities[entity_id]
        else:
            return None

    def process(self, time_passed):
        time_passed_seconds = time_passed / 1000.0
        for entity in self.entities.values():
            entity.process(time_passed_seconds)

    def render(self, surface):
        surface.blit(self.background, (0, 0))
        for entity in self.entities.itervalues():
            entity.render(surface)

    def get_close_entity(self, name, location, range=100.):
        location = Vector2(*location)
        for entity in self.entities.itervalues():
            if entity.name == name:
                distance = location.get_distance_to(entity.location)
                if distance < range:
                    return entity
        return None
```

Monday, May 11, 2009
class Leaf(GameEntity):
    def __init__(self, world, image):
        GameEntity.__init__(self, world, "leaf", image)

class Spider(GameEntity):
    def __init__(self, world, image):
        GameEntity.__init__(self, world, "spider", image)
        self.dead_image = pygame.transform.flip(image, 0, 1)
        self.health = 25
        self.speed = 50. + randint(-20, 20)

    def bitten(self):
        self.health -= 1
        if self.health <= 0:
            self.speed = 0.
            self.image = self.dead_image
            self.speed = 140.

    def render(self, surface):
        GameEntity.render(self, surface)
        x, y = self.location
        w, h = self.image.get_size()
        bar_x = x - 12
        bar_y = y + h/2
        surface.fill((255, 0, 0), (bar_x, bar_y, 25, 4))
        surface.fill((0, 255, 0), (bar_x, bar_y, self.health, 4))

    def process(self, time_passed):
        x, y = self.location
        if x > SCREEN_SIZE[0] + 2:
            self.world.remove_entity(self)
            return
        GameEntity.process(self, time_passed)
class Ant(GameEntity):
    def __init__(self, world, image):
        GameEntity.__init__(self, world, "ant", image)
        exploring_state = AntStateExploring(self)
        seeking_state = AntStateSeeking(self)
        delivering_state = AntStateDelivering(self)
        hunting_state = AntStateHunting(self)

        self.brain.add_state(exploring_state)
        self.brain.add_state(seeking_state)
        self.brain.add_state(delivering_state)
        self.brain.add_state(hunting_state)
        self.carry_image = None

    def carry(self, image):
        self.carry_image = image

    def drop(self, surface):
        if self.carry_image:
            x, y = self.location
            w, h = self.carry_image.get_size()
            surface.blit(self.carry_image, (x-w, y-h/2))
            self.carry_image = None

    def render(self, surface):
        GameEntity.render(self, surface)
        if self.carry_image:
            x, y = self.location
            w, h = self.carry_image.get_size()
            surface.blit(self.carry_image, (x-w, y-h/2))
class AntStateExploring(State):
    def __init__(self, ant):
        State.__init__(self, "exploring")
        self.ant = ant

    def random_destination(self):
        w, h = SCREEN_SIZE
        self.ant.destination = Vector2(randint(0, w), randint(0, h))

    def do_actions(self):
        if randint(1, 20) == 1:
            self.random_destination()

    def check_conditions(self):
        leaf = self.ant.world.get_close_entity("leaf", self.ant.location)
        if leaf is not None:
            self.ant.leaf_id = leaf.id
            return "seeking"
        spider = self.ant.world.get_close_entity("spider", NEST_POSITION, NEST_SIZE)
        if spider is not None:
            if self.ant.location.get_distance_to(spider.location) < 100.:
                self.ant.spider_id = spider.id
                return "hunting"
            return None

    def entry_actions(self):
        self.ant.speed = 120. + randint(-30, 30)
        self.random_destination()
class AntStateSeeking(State):
    def __init__(self, ant):
        State.__init__(self, "seeking")
        self.ant = ant
        self.leaf_id = None

    def check_conditions(self):
        leaf = self.ant.world.get(self.ant.leaf_id)
        if leaf is None:
            return "exploring"
        if self.ant.location.get_distance_to(leaf.location) < 5.0:
            self.ant.carry(leaf.image)
            self.ant.world.remove_entity(leaf)
            return "delivering"
        return None

    def entry_actions(self):
        leaf = self.ant.world.get(self.ant.leaf_id)
        if leaf is not None:
            self.ant.destination = leaf.location
            self.ant.speed = 100. + randint(-20, 20)
class AntStateDelivering(State):
    def __init__(self, ant):
        State.__init__(self, "delivering")
        self.ant = ant

    def check_conditions(self):
        if Vector2(*NEST_POSITION).get_distance_to(self.ant.location) < NEST_SIZE:
            if randint(1, 10) == 1:
                self.ant.drop(self.ant.world.background)
        return "exploring"

    def entry_actions(self):
        self.ant.speed = 60.
        random_offset = Vector2(randint(-20, 20), randint(-20, 20))
        self.ant.destination = Vector2(*NEST_POSITION) + random_offset
class AntStateHunting(State):
    def __init__(self, ant):
        State.__init__(self, "hunting")
        self.ant = ant
        self.got_kill = False

    def do_actions(self):
        spider = self.ant.world.get(self.ant.spider_id)
        if spider is None:
            return
        self.ant.destination = spider.location
        if self.ant.location.get_distance_to(spider.location) < 15.:
            if randint(1, 5) == 1:
                spider.bitten()
            if spider.health <= 0:
                self.ant.carry(spider.image)
                self.ant.world.remove_entity(spider)
                self.got_kill = True
        return None

    def check_conditions(self):
        if self.got_kill:
            return "delivering"
        spider = self.ant.world.get(self.ant.spider_id)
        if spider is None:
            return "exploring"
        if spider.location.get_distance_to(NEST_POSITION) > NEST_SIZE * 3:
            return "exploring"
        return None

    def entry_actions(self):
        self.speed = 160. + randint(0, 50)

    def exit_actions(self):
        self.got_kill = False
def run():
    pygame.init()
    screen = pygame.display.set_mode(SCREEN_SIZE, 0, 32)
    world = World()
    w, h = SCREEN_SIZE
    clock = pygame.time.Clock()
    ant_image = pygame.image.load("ant.png").convert_alpha()
    leaf_image = pygame.image.load("leaf.png").convert_alpha()
    spider_image = pygame.image.load("spider.png").convert_alpha()
    for ant_no in xrange(ANT_COUNT):
        ant = Ant(world, ant_image)
        ant.location = Vector2(randint(0, w), randint(0, h))
        ant.brain.set_state("exploring")
        world.add_entity(ant)

    while True:
        for event in pygame.event.get():
            if event.type == QUIT:
                return
        time_passed = clock.tick(30)
        if randint(1, 10) == 1:
            leaf = Leaf(world, leaf_image)
            leaf.location = Vector2(randint(0, w), randint(0, h))
            world.add_entity(leaf)
        if randint(1, 100) == 1:
            spider = Spider(world, spider_image)
            spider.location = Vector2(-50, randint(0, h))
            spider.destination = Vector2(w+50, randint(0, h))
            world.add_entity(spider)
            world.process(time_passed)
            world.render(screen)
            pygame.display.update()

if __name__ == "__main__":
    run()}