Multimedia-Programmierung
Übung 9

Ludwig-Maximilians-Universität München

Sommersemester 2011

Today

- State Machines in pygame

AI in Games

- Intelligent behavior (e.g. decision making) makes characters in games more realistic
- AI in games: decide on current knowledge and state, which steps to take next
- Examples: Enemy only attacks player in certain range, Sims decide on their next activity based on current mood

State Machines

consist of:

- states
- start state

- state actions
- entry and exit actions

- transitions
- transition conditions

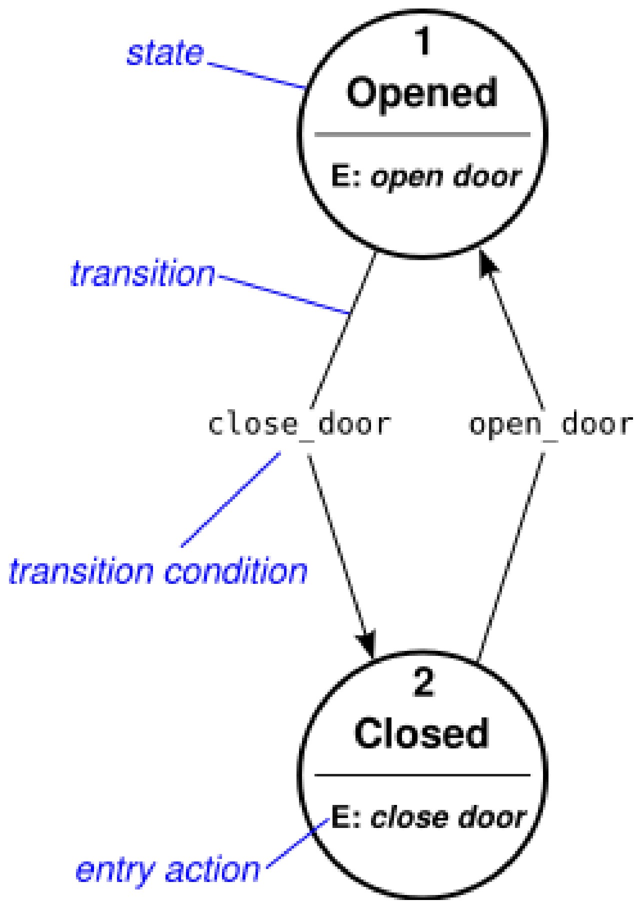
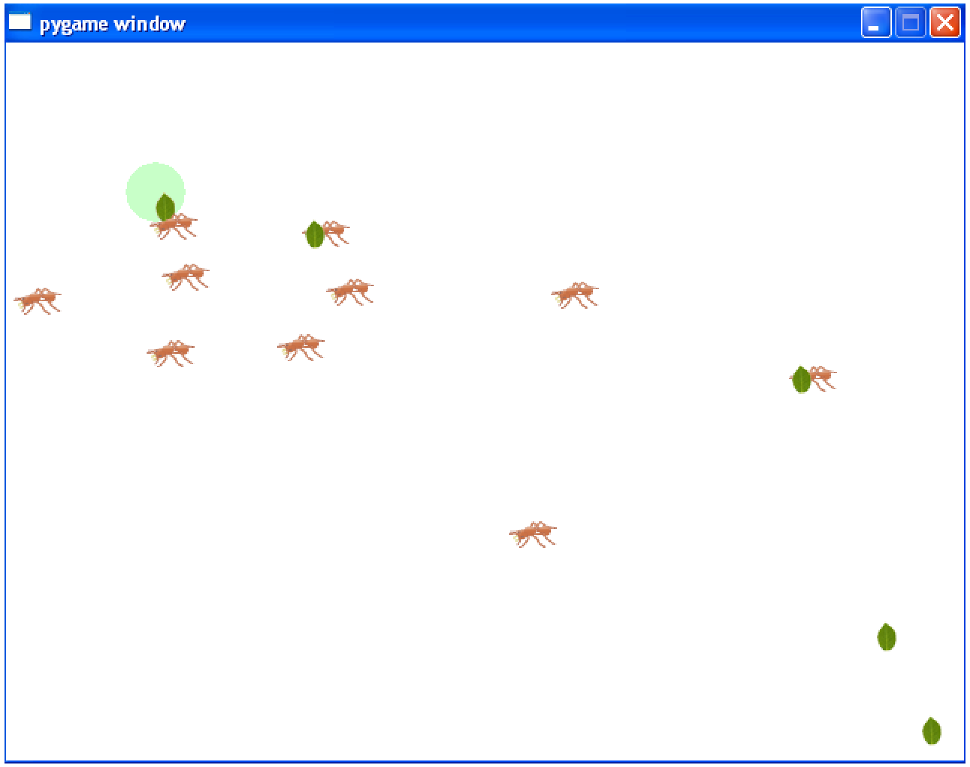


image source: Wikipedia

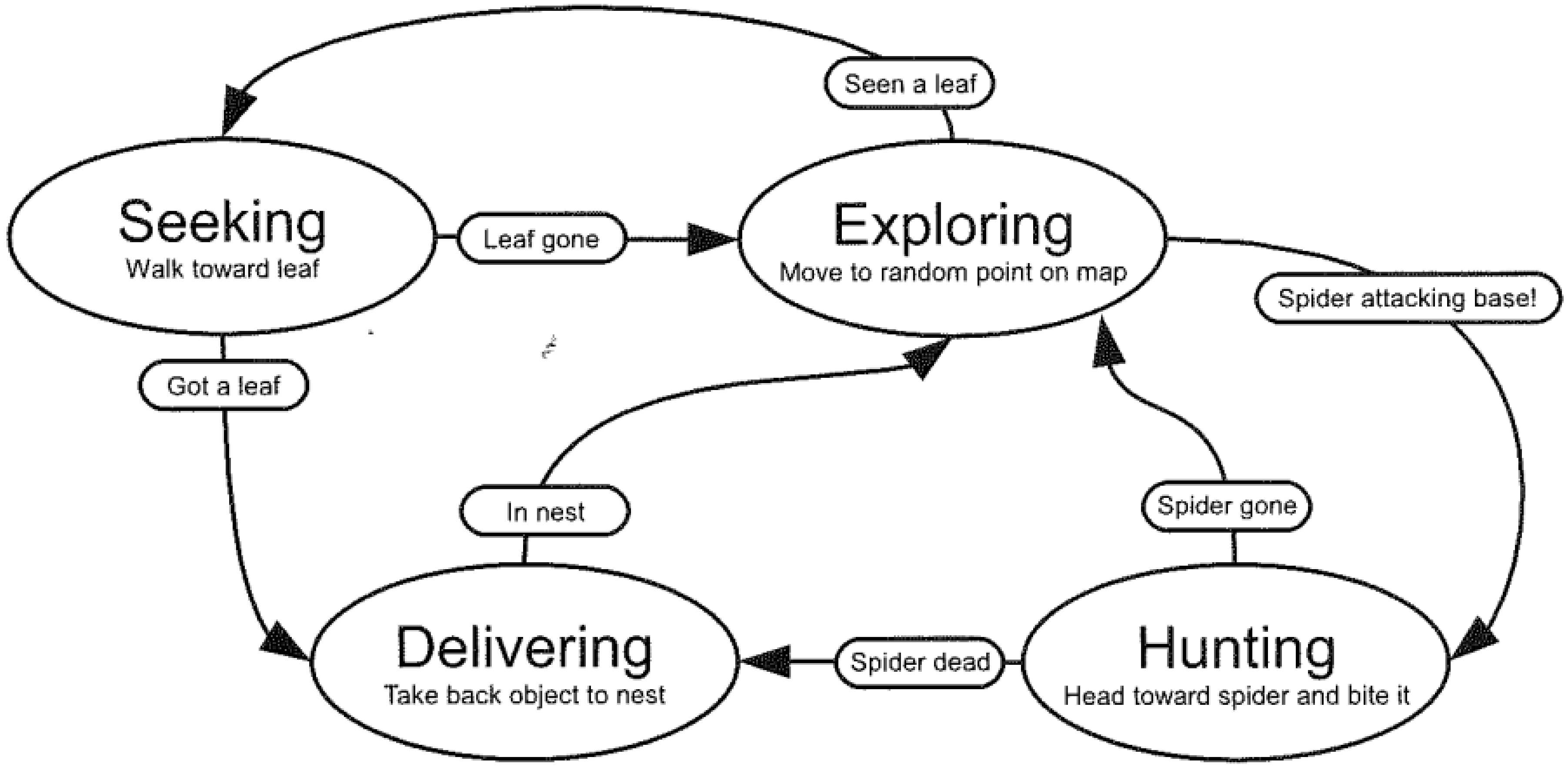
Example: Ant Nest

- Ants search for food and deliver it to their nest



Example from Book „Beginning Game Development with Python and Pygame – From Novice to Professional“ by Will McGugan

Example: Ant Nest



State Machine

```
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()
```

List of states

is called in every update(..)

1. Do actions for current state
2. Check if state changed
3. Eventually change state

4. Do exit actions for old state
5. Do entry actions for new state

State

```
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
```

- `do_actions`: Actions in this state (e.g. update animation, walk somewhere etc.)
- `check_conditions`: Check conditions for this state and eventually change to another state
- `entry_actions`: If changed to this state, do specific actions
- `exit_actions`: If current state gets inactive, do some exit actions

State Example

```
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
```

```
import random
class AntStateExploring(State):
    def __init__(self, ant):
        State.__init__(self, "exploring")
        self.ant = ant

    def do_actions(self):
        #change direction in approx. every 20th call
        if random.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"
        return None

    def entry_actions(self):
        self.ant.speed = 120. + random.randint(-30,30)
        self.random_destination()

    def random_destination(self):
        …
```

Other useful classes for game development

BaseClass for **Game Entities**:

- Moving the game entity
- Rendering the game entity
- Updating current state
- Etc.

```
class GameEntity(object):
    def __init__(self, world, name, image, initial_position):
        self.world = world
        self.name = name
        self.image = image
        self.location = initial_position
        self.destination = (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()
#calculate new position and move game entity
        …
```

Other useful classes for game development

World:

- Stores all game entities (e.g. in a dictionary) and assigns IDs to new entities
- Starts update and rendering process of entities
- Can provide queries for entities (e.g. entities in range etc.)

```
class World(object):
    def __init__(self):
        self.entities = {}
        self.entity_id = 0
self.background = …

    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):
        …
```

Other useful classes for game development

```
…
    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.values():
            entity.render(surface)

    def get_close_entity(self, name, location, range=100):
        …
```