UNIVERSITÀ DEGLI STUDI
DI MILANO

Finite State Machines

A.I. for Video Games

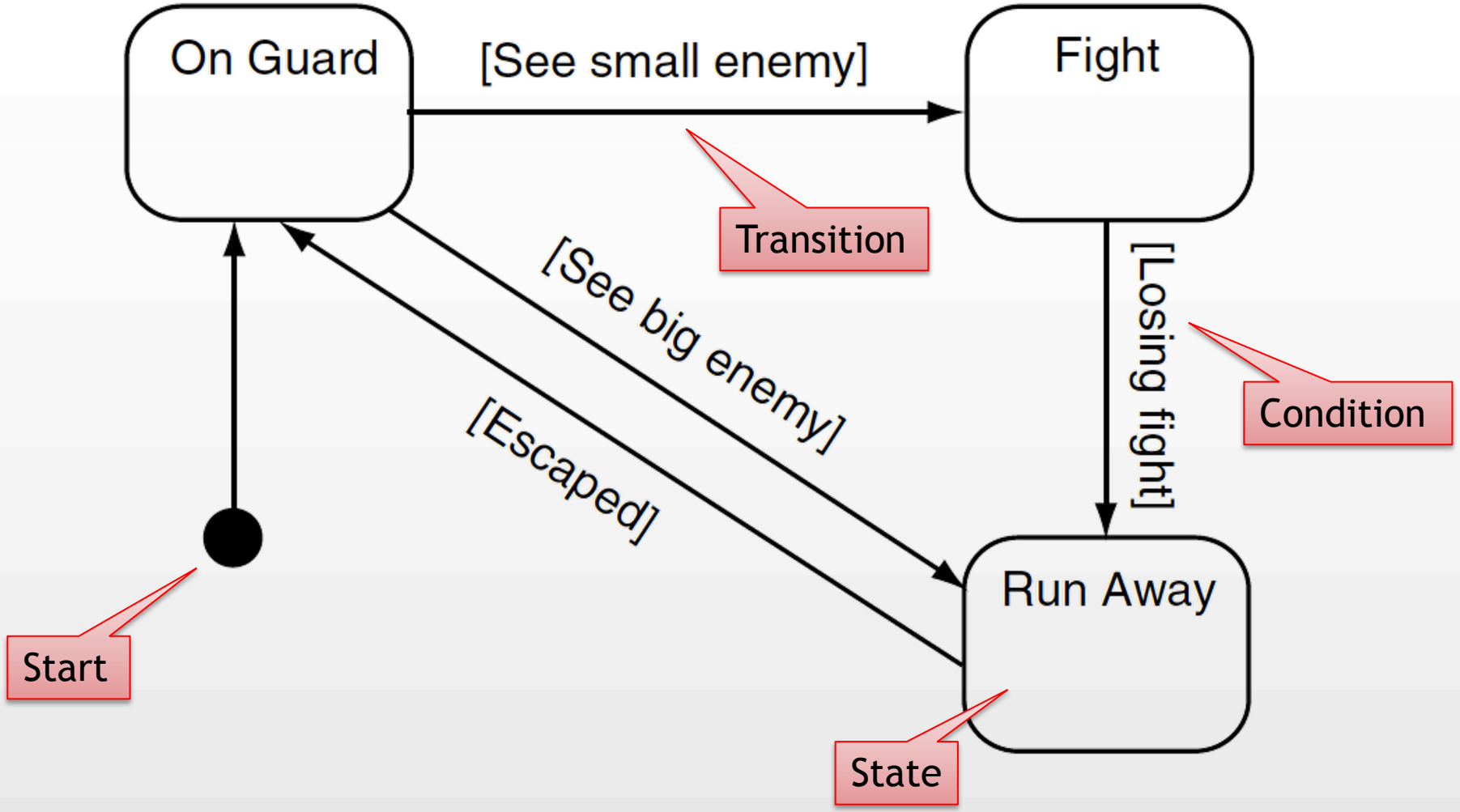
Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

Playlab fOr inNovation in Games

Finite State Machines

- Very often, an NPC must switch between a set of given states (conditions)
  - Each state is significative of a specific condition or activity
- Once in a state, the NPC keeps doing the same thing until an event occurs
  - Then, it will move to another state and iterate this process
- We could achieve the same result with decision trees, but state machines are just simpler
  - And MUCH easier to design

A Sample FSM


On Guard
[See small enemy]
Fight
Transition
[See big enemy]
[Losing fight]
Condition
[Escaped]
Run Away
Start
State

In a Nutshell

- Each agent is in a given state
- NPC available actions and behaviors are associated to each state
- As long as the agent remains in that state, it will continue carrying out the same set of actions
  - NOTE: “action” might be “walk a specific decision tree”
- States are linked by transitions
- Each transition has a set of associated conditions
- If the conditions of a transition are met, then the NPC changes state to the transition’s target state

(Formal) Finite State Machines

- A finite set of states (Q)
- A finite set of input symbols called the alphabet (Σ)
- A transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- An initial or start state ($q0 \in Q$)
- A set of accept states ($F \subseteq Q$)

Yes, I know this is boring ...
but you must remember it for this exam

(Game) Finite State Machines

- A finite set of states (Q)
- ~~A finite set of input symbols called the alphabet (Σ)~~
- A transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- An initial or start state ($q0 \in Q$)
- A set of accept states ($F \subseteq Q$)

FSM **SCAN** the environment, you are NOT providing any input

(Game) Finite State Machines

- A finite set of states (Q)
- ~~A finite set of input symbols called the alphabet (Σ)~~
- ~~A transition function ($\delta : Q \times \Sigma \rightarrow Q$)~~
- An initial or start state ($q0 \in Q$)
- A set of accept states ($F \subseteq Q$)

Therefore, if there are no input symbols, we have no use for a transition function

(Game) Finite State Machines

- A finite set of states (Q)
- ~~A finite set of input symbols called the alphabet (Σ)~~
- ~~A transition function ($\delta : Q \times \Sigma \rightarrow Q$)~~
- Sensors to trigger transitions
- An initial or start state ($q0 \in Q$)
- A set of accept states ($F \subseteq Q$)

The transition function is substituted by sensors triggering transitions

(Game) Finite State Machines

- A finite set of states (Q)
- ~~A finite set of input symbols called the alphabet (Σ)~~
- ~~A transition function ($\delta : Q \times \Sigma \rightarrow Q$)~~
- Sensors to trigger transitions
- An initial or start state ($q0 \in Q$)
- ~~A set of accept states ($F \subseteq Q$)~~

And, since we are not recognizing any input, we do not need to label any state as “final”

Actually, many NPCs are designed to run forever

How Do We Use an FSM?

- We call the state machine's update function periodically

- The update function will:
  - Check to see if there are enabled conditions
  - Schedule the first match to fire its transition
  - Perform actions
  - Fire the transition if one is triggered

It is possible to have more than one!

- It must be also possible to assign (one or more) actions to:
  - Entering a state
  - Firing a transition
  - Exiting a state
  - Staying in a state

Trigger

No trigger

Implementing Transitions

Source: FSM
Folder: FSM

```
// Defer function to trigger activation condition
// Returns true when transition can fire
public delegate bool FSMCondition();

// Defer function to perform action
public delegate void FSMAction();

public class FSMTransition {

    // The method to evaluate if the transition is ready to fire
    public FSMCondition myCondition;

    // A list of actions to perform when this transition fires
    private List<FSMAction> myActions = new List<FSMAction>();

    public FSMTransition(FSMCondition condition, FSMAction[] actions = null) {
        myCondition = condition;
        if (actions != null) myActions.AddRange(actions);
    }

    // Call all  actions
    public void Fire() {
        if(myActions != null) foreach (FSMAction action in myActions) action();
    }
}
```

We keep using delegates to hook code from outside the library

A transition is made of a triggering condition and a list of actions to be performed upon firing

We provide all the actions on instancing by an optional array

On firing, we just call all the actions in the list

Implementing States

Source: FSM
Folder: FSM

A state has three lists of actions as we saw before

This dictionary is linking to new states through transitions

Verify if there are transitions ready to fire. First match is good for us

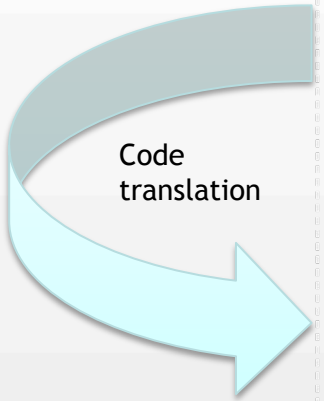
Return the next state given a transition

Execute all actions in the various situations

```
public class FSMState {

    // Arrays of actions to perform based on transitions fire (or not)
    // Getters and setters are preferable, but we want to keep the source clean
    public List<FSMAction> enterActions = new List<FSMAction> ();
    public List<FSMAction> stayActions = new List<FSMAction> ();
    public List<FSMAction> exitActions = new List<FSMAction> ();

    // A dictionary of transitions and the states they are leading to
    private Dictionary<FSMTransition, FSMState> links;

    public FSMState() {
        links = new Dictionary<FSMTransition, FSMState>();
    }

    public void AddTransition(FSMTransition transition, FSMState target) {
        links [transition] = target;
    }

    public FSMTransition VerifyTransitions() {
        foreach (FSMTransition t in links.Keys) {
            if (t.myCondition()) return t;
        }
        return null;
    }

    public FSMState NextState(FSMTransition t) {
        return links [t];
    }

    // These methods will perform the actions in each list
    public void Enter() { foreach (FSMAction a in enterActions) a(); }
    public void Stay() { foreach (FSMAction a in stayActions) a(); }
    public void Exit() { foreach (FSMAction a in exitActions) a(); }

}
```

Implementing an FSM

Source: FSM
Folder: FSM




```
public class FSM {

    // Current state
    public FSMState current;

    public FSM(FSMState state) {
        current = state;
        current.Enter();
    }

    // Examine transitions leading out from the current state
    // If a condition is activated, then:
    // (1) Execute actions associated to exit from the current state
    // (2) Execute actions associated to the firing transition
    // (3) Retrieve the new state and set is as the current one
    // (4) Execute actions associated to entering the new current state
    // Otherwise, if no condition is activated,
    // (5) Execute actions associated to staying into the current state

    public void Update() { // NOTE: this is NOT a MonoBehaviour
        FSMTransition transition = current.VerifyTransitions ();
        if (transition != null) {
            current.Exit();     // 1
            transition.Fire();  // 2
            current = current.NextState(transition);    // 3
            current.Enter();    // 4
        } else {
            current.Stay();     // 5
        }
    }
}
```

FSM Complexity

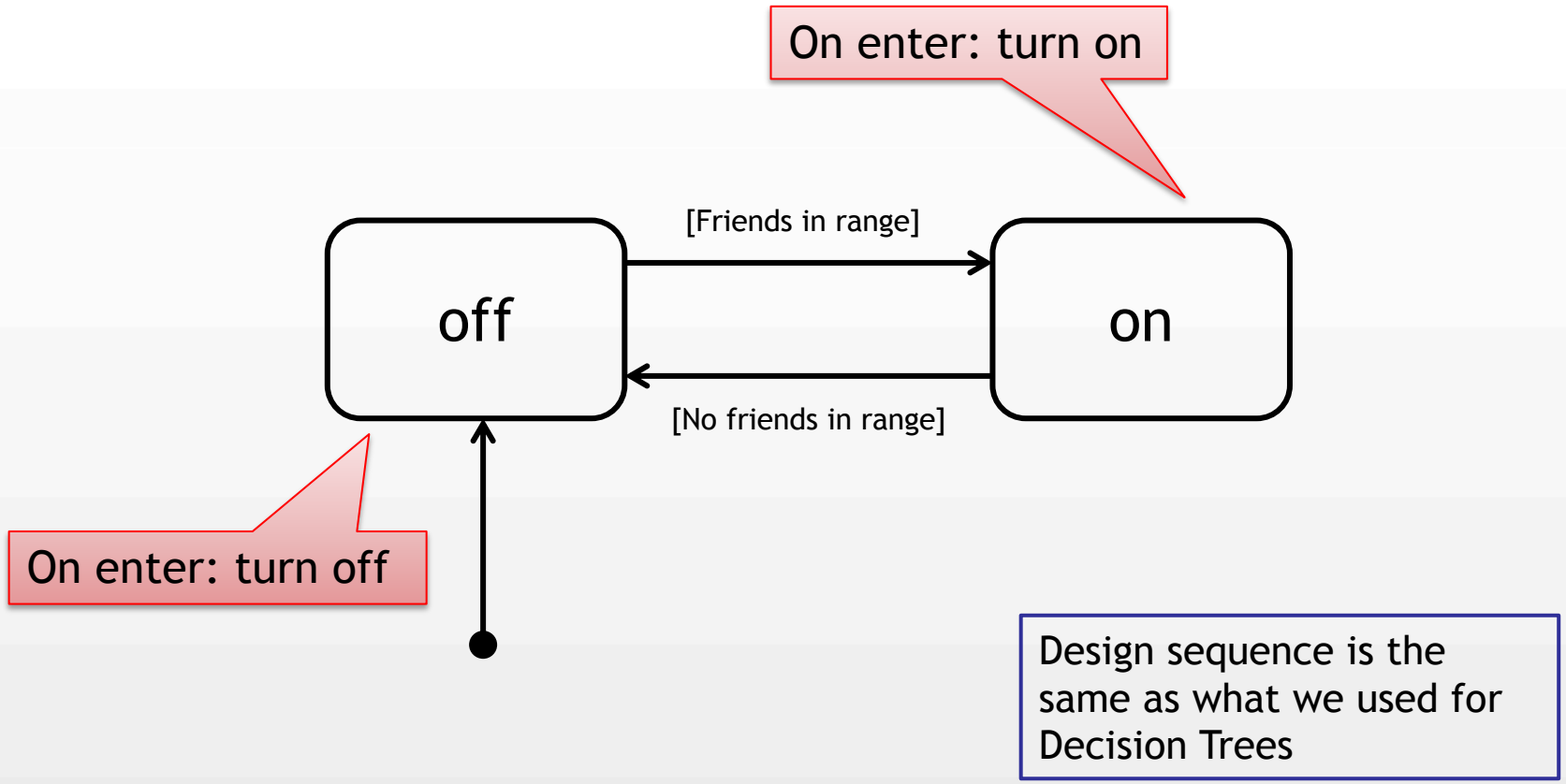
- Good news: it is always linear
  - We always start from the current state → O(1)
  - We scan at most *n* outgoing link → O(n)

- Whatever we do with an FSM we can also do with decision trees (!)
  - Just … FSM are faster because a DT must trace back the current state using internal and external knowledge. Instead, an FSM has memory of the situation (the current state)

On/Off Sentinel

- A sentinel is guarding a dark place
- If a friendly agent comes into range, the sentinel switches on the light
- When the friendly agent leaves, the light is turned off

On/Off Sentinel



Design sequence is the same as what we used for Decision Trees

On/Off Sentinel

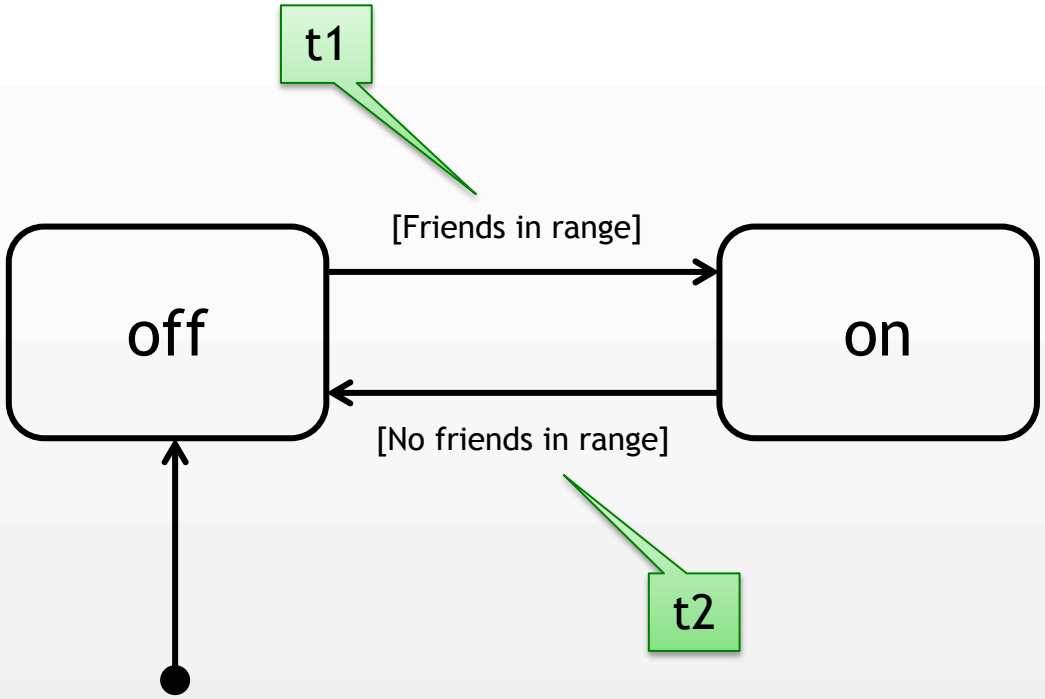
Source: OnOffSentinel
Folder: FSM

```
// CONDITIONS

public bool FriendsInRange() {
    foreach (GameObject go in GameObject.FindGameObjectsWithTag(targetTag)) {
        if ((go.transform.position - transform.position).magnitude <= range) return true;
    }
    return false;
}

public bool NoFriendsInRange() {
    return !FriendsInRange();
}

// ACTIONS

public void TurnOn() {
    ambientLight.intensity = 1f;
}

public void TurnOff() {
    ambientLight.intensity = 0f;
}
```

We need to reverse logic here

This time we are not changing light color but its intensity

On/Off Sentinel



On/Off Sentinel

```
[Range(0f, 20f)] public float range = 5f;
public float reactionTime = 3f;
public string targetTag = "Player";
public Light ambientLight = null;

private FSM fsm;

void Start () {
    if (!ambientLight) return;  // Sanity

    // Define states and add actions when enter/exit/stay
    FSMState off = new FSMState();
    off.enterActions.Add(TurnOff);

    FSMState on = new FSMState();
    on.enterActions.Add(TurnOn);

    // Define transitions
    FSMTransition t1 = new FSMTransition (FriendsInRange);
    FSMTransition t2 = new FSMTransition (NoFriendsInRange);

    // Link states with transitions
    off.AddTransition(t1, on);
    on.AddTransition(t2, off);

    // Setup a FSA at initial state
    fsm = new FSM(off);

    // Start monitoring
    StartCoroutine(Patrol());
}
```

Create first state with its associated actions

Create second state with its associated actions

Create transitions (only delegate is needed)

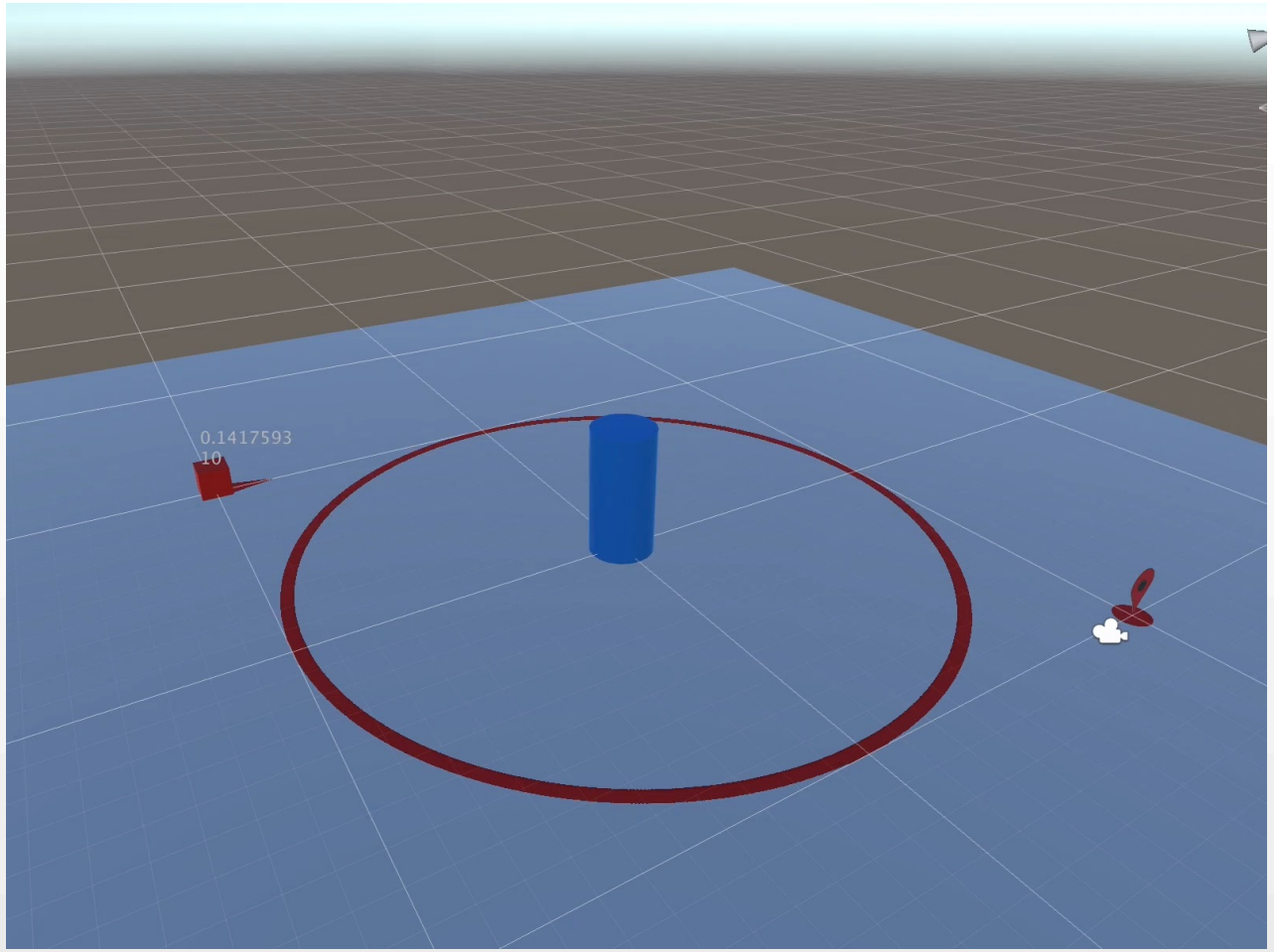
Link states using transitions

Create an FSM starting in state “off”

Last, start a coroutine that keeps updating the FSM every *reactionTime* seconds

Looks Good!

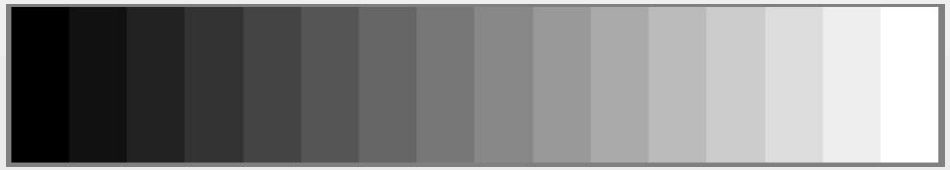
Scene: OnOff Sentinel with FSM
Folder: FSM



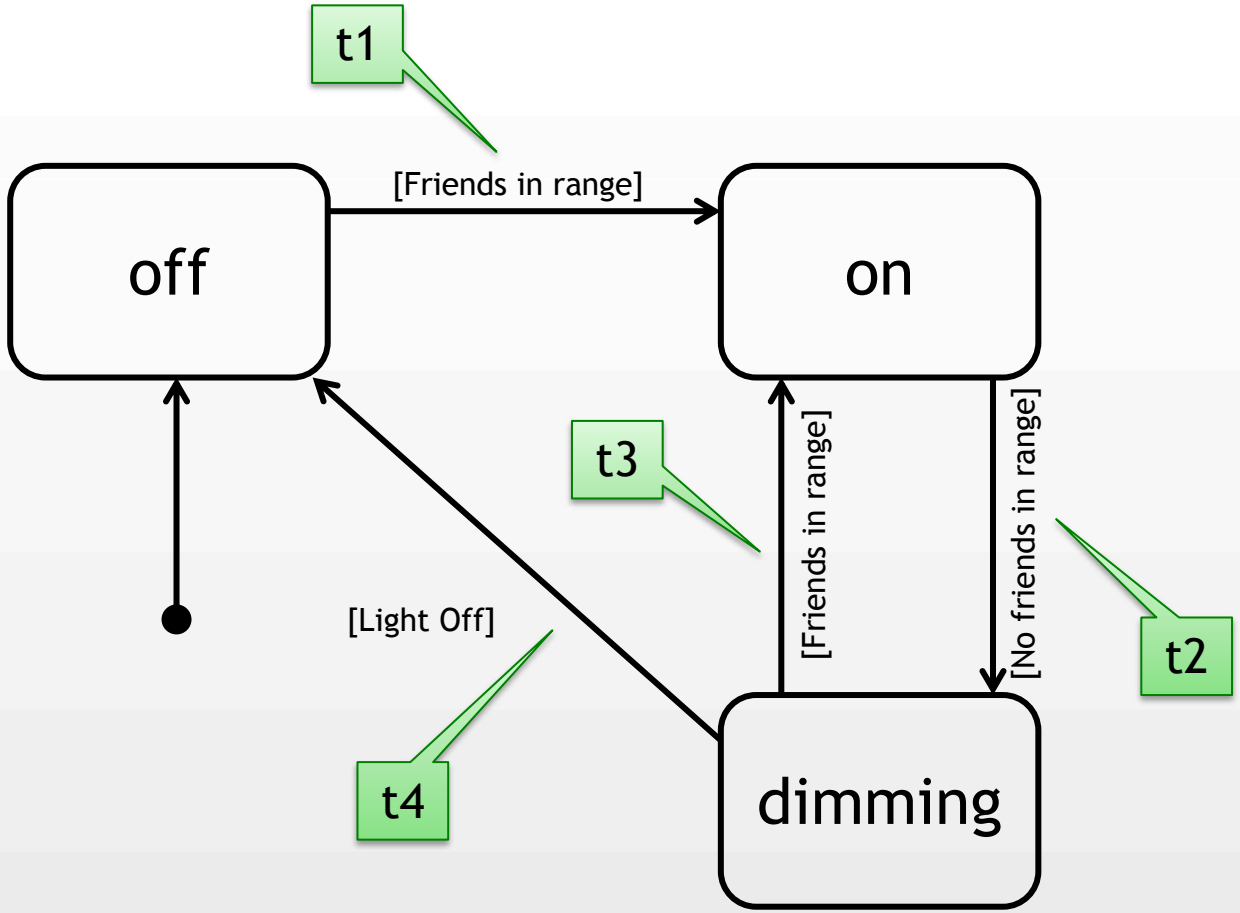
Reaction delay is still bounded by our frame time

Dimming Sentinel

- A sentinel is guarding a dark place
- If a friendly target comes into range, the sentinel switches on the light
- When the friendly target leaves, the light is dimmed off
- If another friendly target comes into range while dimming, the light is switched back on



Dimming Sentinel



Dimming Sentinel

Source: DimmingSentinel
Folder: FSM

```
// CONDITIONS

public bool FriendsInRange() {
    foreach (GameObject go in GameObject.FindGameObjectsWithTag(targetTag)) {
        if ((go.transform.position - transform.position).magnitude <= range) return true;
    }
    return false;
}

public bool NoFriendsInRange() {
    return !FriendsInRange();
}

public bool LightIsOff() {
    return (ambientLight.intensity == 0);
}

// ACTIONS

public void TurnOn() {
    ambientLight.intensity = 1f;
}

public void TurnOff() {
    ambientLight.intensity = 0f;
}

public void StartDimmer () {
    dimmingStart = Time.realtimeSinceStartup;
}

public void Dimmer() {
    ambientLight.intensity = 1f - Mathf.Clamp01 ((Time.realtimeSinceStartup - dimmingStart) / dimmingTime);
}
```

Red boxes outline the new stuff.
All the other code is the same as in the previous example

Dimming Sentinel

```
// CONDITIONS

public bool FriendsInRange() {
    foreach (GameObject go in GameObject.FindGameObjectsWithTag(targetTag)) {
        if ((go.transform.position - transform.position).magnitude <= range) return true;
    }
    return false;
}

public bool NoFriendsInRange() {
    return !FriendsInRange();
}

public bool LightIsOff() {
    return (ambientLight.intensity == 0);
}

// ACTIONS

public void TurnOn() {
    ambientLight.intensity = 1f;
}

public void TurnOff() {
    ambientLight.intensity = 0f;
}

public void StartDimmer () {
    dimmingStart = Time.realtimeSinceStartup;
}

public void Dimmer() {
    ambientLight.intensity = 1f - Mathf.Clamp01 ((Time.realtimeSinceStartup - dimmingStart) / dimmingTime);
}
```

We must check if the light is off to go back to the ”off” state

The visual effect is influenced by the A.I. frame rate (we will not see a gradual dimming unless the frame time is very short.

A coroutine could do a better job but will be more difficult to control if we have to interrupt the dimming (action “TurnOn”). This is because a coroutine is oblivious of the FSM state

Start dimming means marking the timestamp. Current intensity (frame by frame) will be calculated while staying in the “dimming” state

While we stay in the dimming state, we calculate and set the light intensity

Dimming Sentinel

```
void Start () {
    if (!ambientLight) return;  // Sanity

    // Define states and link actions when enter/exit/stay
    FSMState off = new FSMState ();
    off.enterActions.Add (TurnOff);

    FSMState on = new FSMState();
    on.enterActions.Add (TurnOn);

    FSMState dimming = new FSMState();
    dimming.enterActions.Add (StartDimmer);
    dimming.stayActions.Add (Dimmer);

    // Define transitions
    FSMTransition t1 = new FSMTransition (FriendsInRange);
    FSMTransition t2 = new FSMTransition (NoFriendsInRange);
    FSMTransition t3 = new FSMTransition (FriendsInRange); // different from t1
    FSMTransition t4 = new FSMTransition (LightIsOff);

    // Link states with transitions
    off.AddTransition (t1, on);
    on.AddTransition (t2, dimming);
    dimming.AddTransition (t3, on);
    dimming.AddTransition (t4, off);

    // Setup a FSA at initial state
    fsm = new FSM(off);

    // Start monitoring
    StartCoroutine(Patrol());
}
```

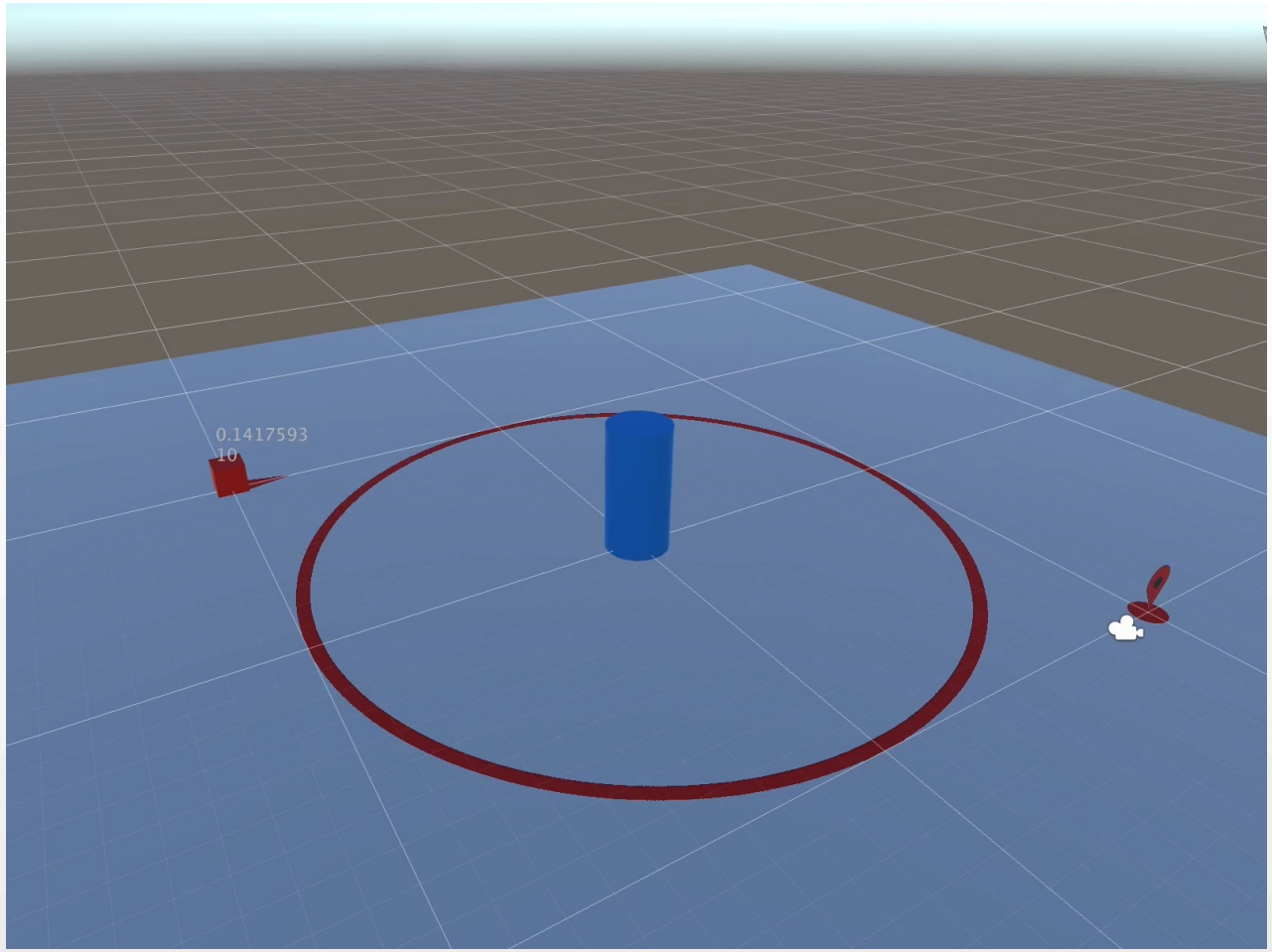
FSM setup is basically the same as before. We just have more states and transitions to declare

Transitions t1 and t3 are NOT the same because they are used in different contexts (between different states).

Code wise, of course, you can use only one instance, but the code will be much less manageable.
If your designer will change the logic for only one of the transitions your build will break!

Dimming Out

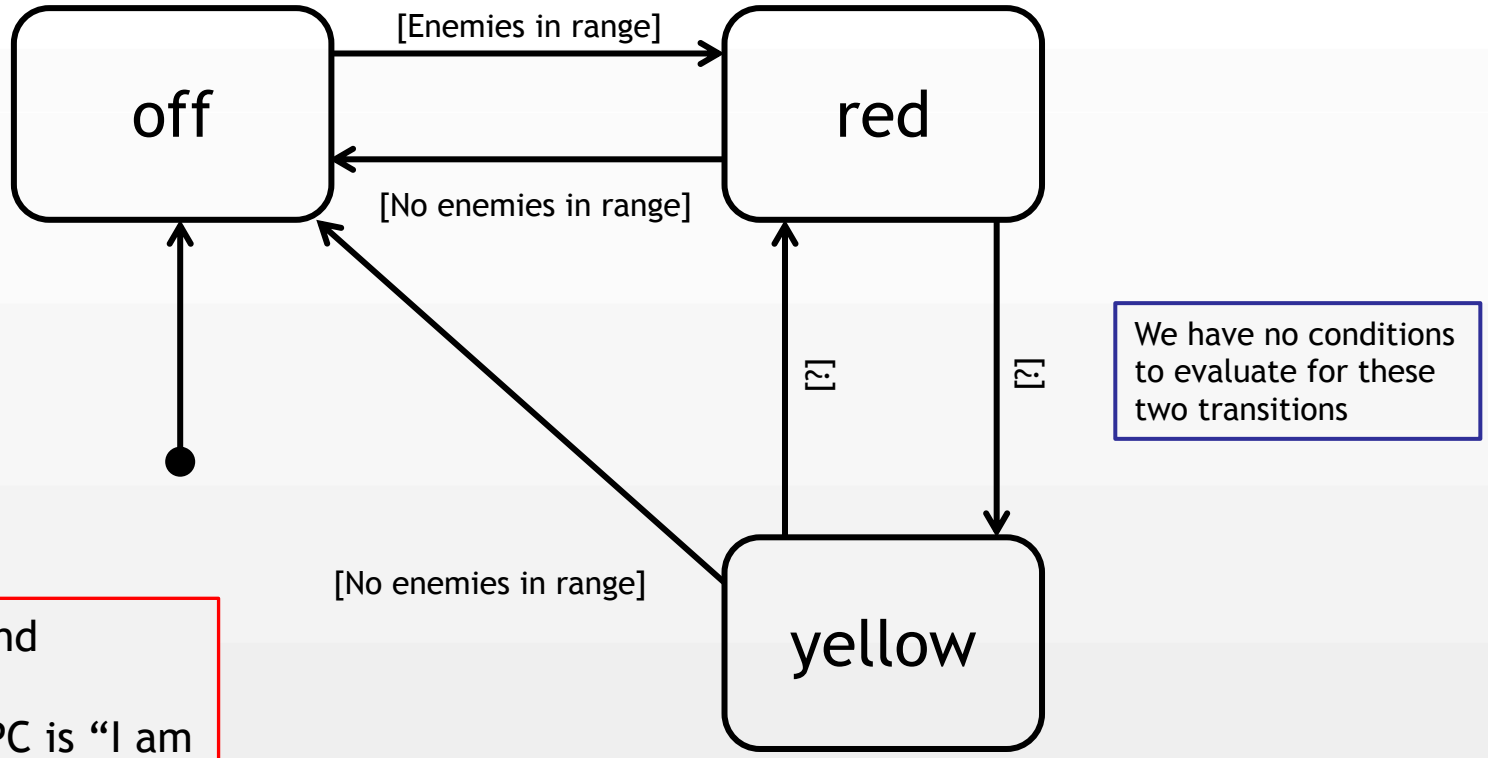
Scene: Dimming Sentinel with FSM
Folder: FSM



Alarm Sentinel

- A sentinel is guarding a dark place
- If an enemy comes into range, the sentinel calls the alarm
  - The alarm is a red/yellow light alternation
- When the enemy leaves, the alarm is turned off

Alarm Sentinel … the Wrong Way



We have no conditions to evaluate for these two transitions

The problem is that “red” and “yellow” are NOT states!
Because the state for the NPC is “I am running the alarm”

Beware of the Error!

- How do I switch between red and yellow?
  - Automatic transitions are fine but hints to redundancy

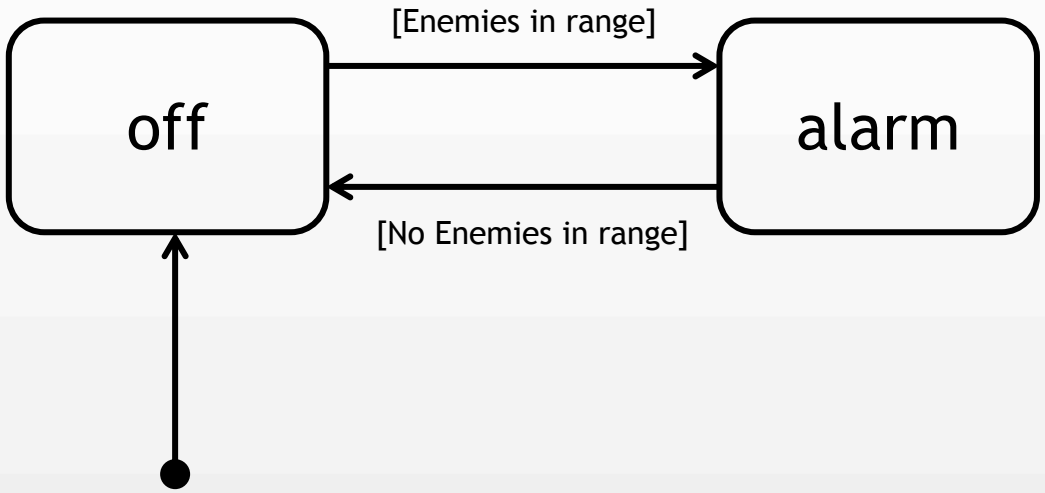
```
public bool AlwaysSwitch() {
    return true;
}
```

This can be used as a delegate, but everyone gets the feeling we are wasting resources

- Alarm semantic is wired inside the FSM schema
  - And is constrained by the A.I. framerate

- What if we change the frame time?
  - Gaming experience will be different
  - Game mechanics might break
    - E.g., if the player need to do something only when the light is red

Alarm Sentinel … a Better Option



Basically, we reverted to the On/Off sentinel schema

Alarm Sentinel

Source: AlarmSentinel
Folder: FSM

```
// CONDITIONS

public bool EnemiesAround() {
    foreach (GameObject go in GameObject.FindGameObjectsWithTag(targetTag)) {
        if ((go.transform.position - transform.position).magnitude <= range) return true;
    }
    return false;
}

public bool NoEnemiesAround() {
    return !EnemiesAround();
}

// ACTIONS

public void StartAlarm () {
    initialColor = ambientLight.color;
    ringStart = Time.realtimeSinceStartup;
}

public void ShutAlarm() {
    ambientLight.color = initialColor;
}

public void RingAlarm() {
    if ((int)Mathf.Floor ((Time.realtimeSinceStartup - ringStart) / switchTime) % 2 == 0) {
        ambientLight.color = color1;
    } else {
        ambientLight.color = color2;
    }
}
```

Conditions are the same as in the I/O sentinel but with a renaming due to the new context

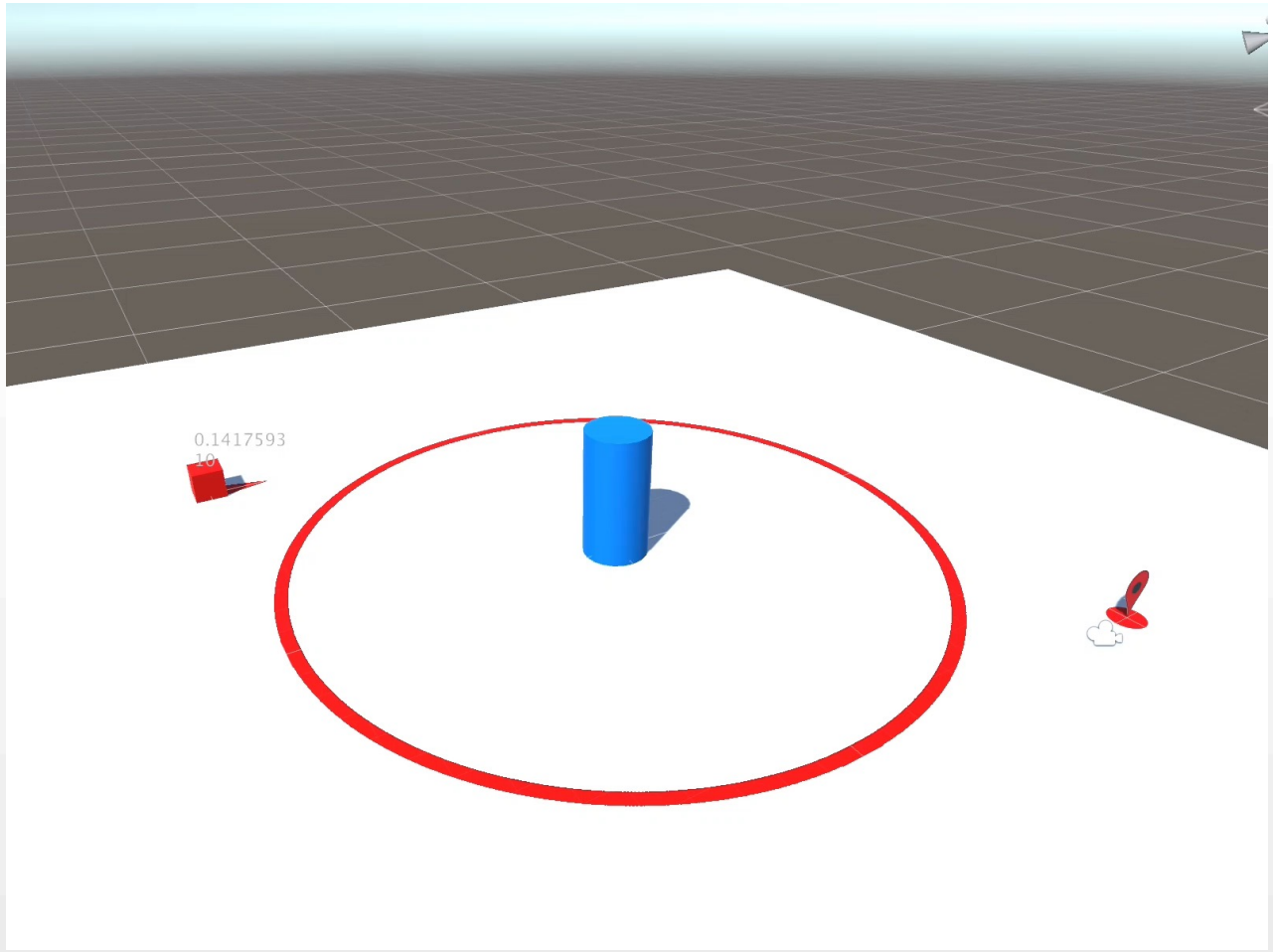
Actions are very similar to the dimming sentinel example.
We do not need to switch off the light and we are alternating color based on seconds (even or odd) from starting the alarm

Alarm Sentinel

```
void Start () {
    if (!ambientLight) return;  // Sanity

    // Define states and link actions when enter/exit/stay
    FSMState off = new FSMState();

    FSMState alarm = new FSMState();
    alarm.enterActions.Add(StartAlarm);
    alarm.stayActions.Add(RingAlarm);
    alarm.exitActions.Add(ShutAlarm);

    // Define transitions
    FSMTransition t1 = new FSMTransition (EnemiesAround);
    FSMTransition t2 = new FSMTransition (NoEnemiesAround);

    // Link states with transitions
    off.AddTransition (t1, alarm);
    alarm.AddTransition (t2, off);

    // Setup a FSA at initial state
    fsm = new FSM(off);

    // Start monitoring
    StartCoroutine(Patrol());
}
```

Ringing the Alarm

Scene: Alarm Sentinel with FSM
Folder: FSM



(Optional) Exercise

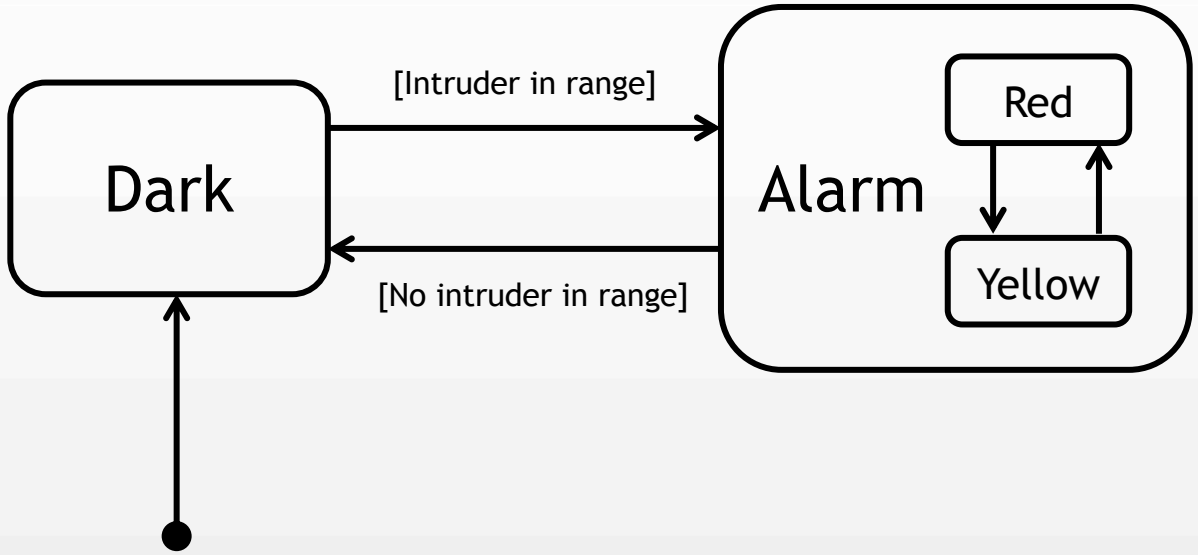
- Create a FSM for an *Alarm Sentinel* which is also dimming
- Hint: beware of the timing!

(Optional) Exercise

- Create a versatile sentinel
  - When one or more friends are in range the light is on (white)
  - When one or more intruders are in range the alarm is on
  - When both friends and intruders are in range the light is on (yellow)

- Hint: once again, beware of the timing!

Now … Think in this Way



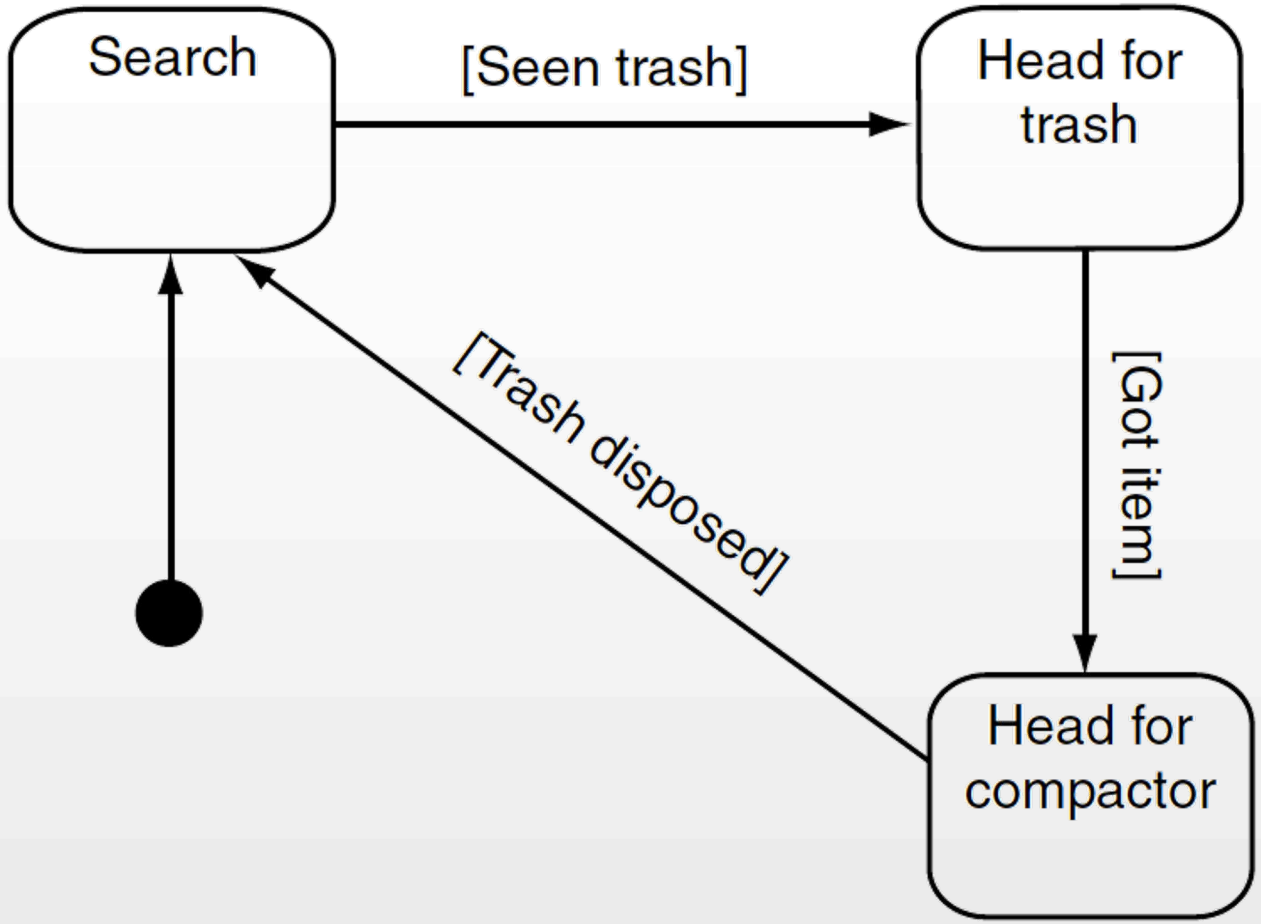
The NPC have an alarm state where, internally, there are two sub-states: “alarm with red light” and “alarm with yellow light”

Alarm Behaviours

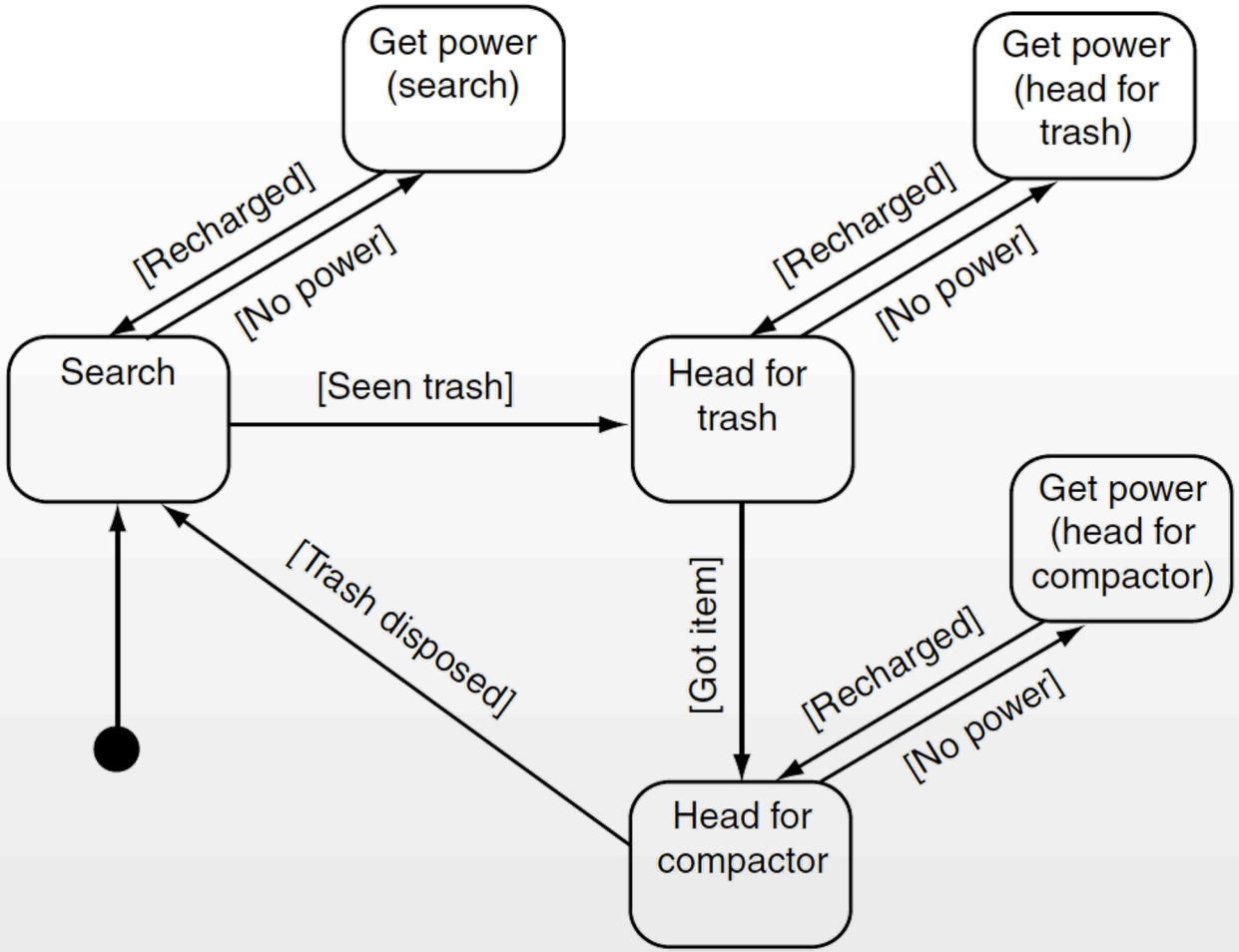
NOTE: This “alarm” has nothing to do with the previous examples. The name is coming from “alarm watch” waking you up in the morning when you are required to stop what you are doing (sleeping) and go to school

- A problem for FSM are alarm behaviours
  - I.e., we have something important to do and we must **suspend** the current FSM
- We can oblige with “standard” FSM, but the solution will be clumsy and difficult to maintain
  - Just think putting IN EVERY STATE of an FSM a transition going to the state (or more than one state) managing the alarm
  - ... and think about doing that for every alarm
  - ... and then, how do you come back to the “right” interrupted state?
- The right answer is a Hierarchical FSM (HFSM)
  - Where we put FSMs one inside anther like a matryoshka

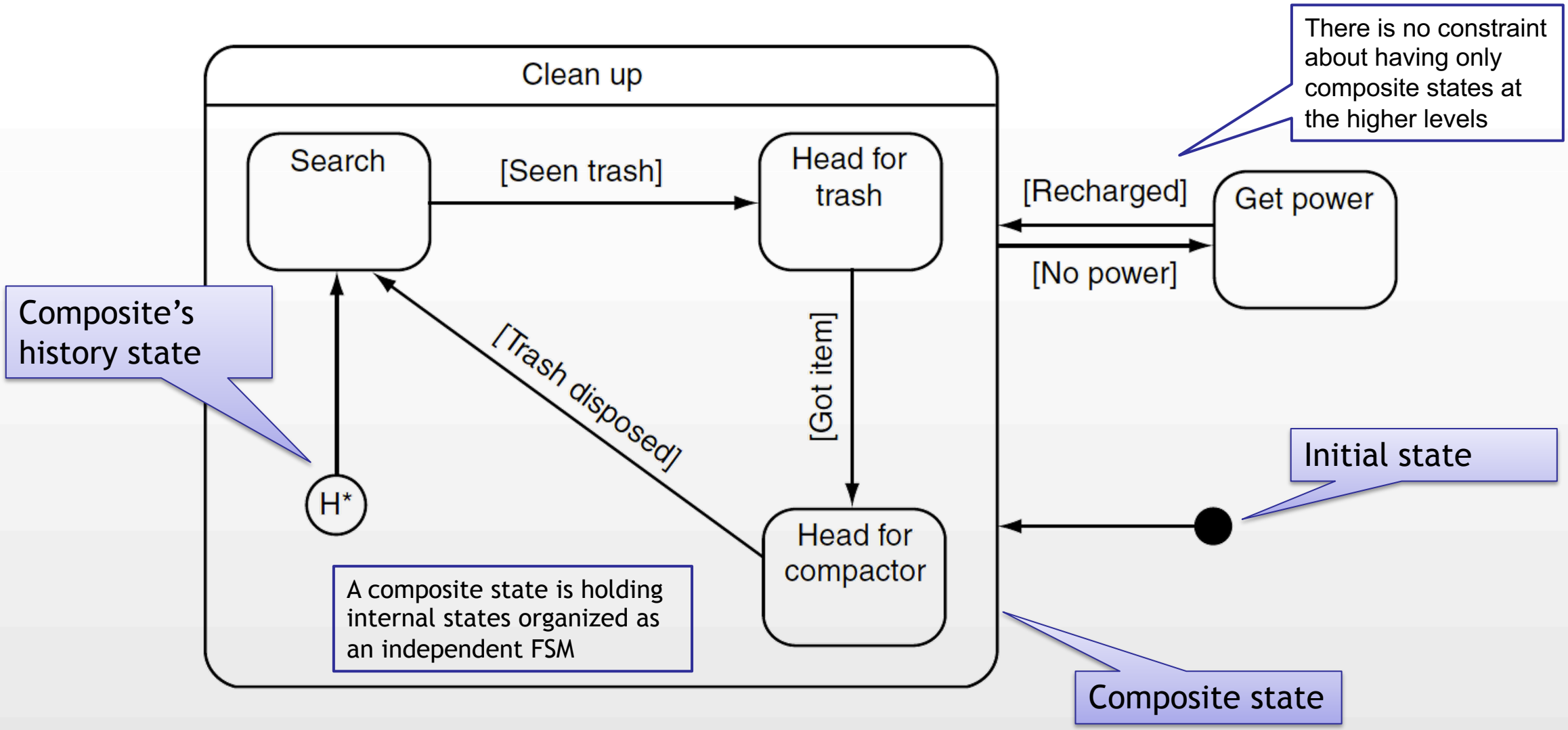
Cleaning Robot (Example from the Book)



Cleaning Robot with Power Management



Cleaning Robot with Power Management



Possible Implementation (not in the Archive)

- We can instantiate TWO FSMs
  - MainFSM
  - CleaningFSM

- In the main loop, we update only the main (outer) FSM

- When we stay in the cleaning state, we can just update the cleaning FSM
  - Quick and dirty: cleaningState.stayActions.Add(CleaningFSM.Update);

Priority

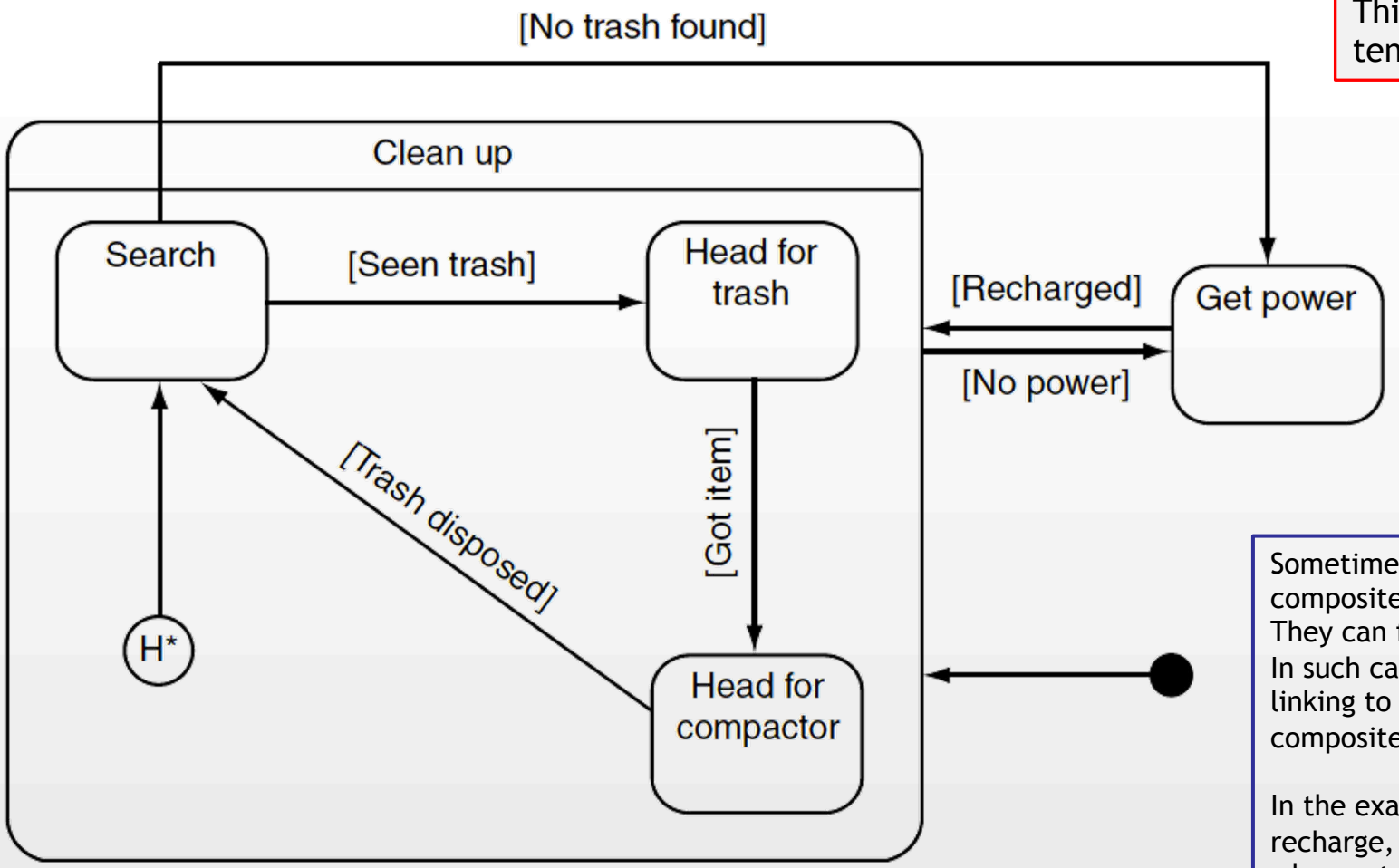
- What if we have multiple transitions ready to fire on different levels of the HFSM?

- Hierarchy *IS* the priority
  - The outer FSM is always more important
  - A higher-level transition will fire first even if there are lower-level condition verified

- You must plan your AI accordingly
  - Usually, this means defining level of abstractions (or divide your AI in simple and reusable tasks)

Exiting From a Composite State

This is breaking our tentative implementation!



Sometimes, transitions to leave a composite state are not always valid. They can fire only in specific sub-state. In such cases, we have transitions linking to states outside the current composite.

In the example, the robot can decide to recharge, even if it still has power, when not finding trash for a while

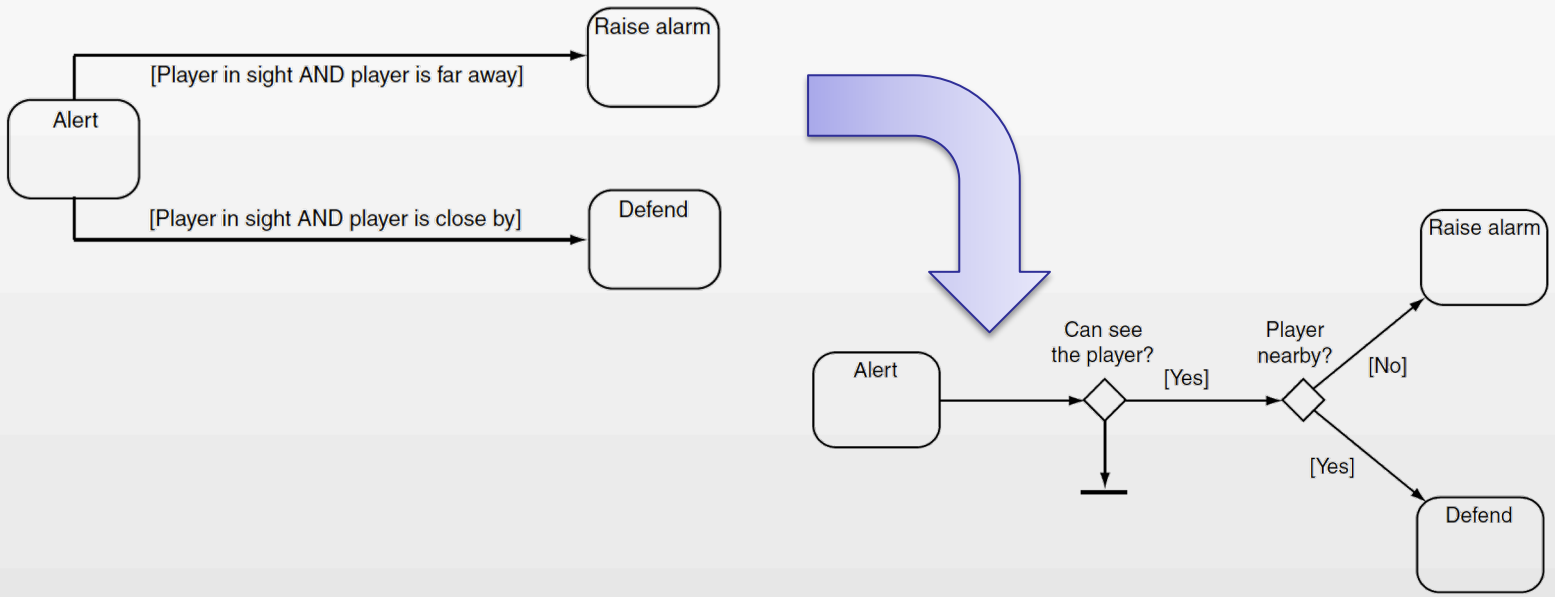
Full Implementation of HFSMs

- This may get a bit messy
- Easy way: set a condition (a variable value?) to force transition at a different level in the next frame
  - Basically you put an option on the future
    - What if a condition with an even higher priority is verified in the next frame?
  - If polling time is long, this is going to be a problem
    - Player will see a sluggish response
  - Formally not clean
    - Because we are breaking the definition of FSM, which is not bound to temporal information
- Complex way:
  - Extend the transition class to address a state in another FSM
  - Rewrite the way an FSM is updated to use the new transition

Check second part of section § 5.3.8 in the book for a more detailed analysis

Combining FSM and DT

- We can combine the two approaches by replacing transitions with a decision tree
  - This can be useful if conditions are very complex and partially overlapped
- The leaves of the tree are transitions to new states
  - Basically, we walk a Decision Tree to understand is there is a valid condition leading out of the current state
  - The leaf we reach is the state whose transition should fire



References

- On the book
  - § 5.3 (excluded 5.3.3, 5.3.4, 5.3.5, 5.3.7)