COMP 4303:
Artificial Intelligence in Games

Decision Making and Finite State Machiens

Jay Henderson (jayhend@mun.ca)

In this section

Decision Making Overview

Finite State Machines

Decision Making

Decision Making

Decision making is the ability of a character/agent to decide what to do

The agent processes information that it uses to generate an action to carry out

Input: agent’s knowledge about the world

Output: an action request

Decision Making: Knowledge

External knowledge: information about the game environment

(e.g. characters' positions, level layout, impending collisions, etc.)

Internal knowledge: information about the character's internal state

(e.g. health, goals, last actions, etc.)

Decision Making

Usually, game characters have a limited set of possible behaviors:
Do the same thing until some event or influence makes them change

e.g. a guard will stand at its post until it notices the player,
then it will switch into attack mode, taking cover and firing

State machines are most often used to implement this

Finite State Machines

What is a Finite State Machine?

A finite state machine, or **FSM** for short, is a model of computation based on a hypothetical machine made of one or more states

Only **one state** can be active at a time, so the machine must **transition** from one state to another in order to perform different actions

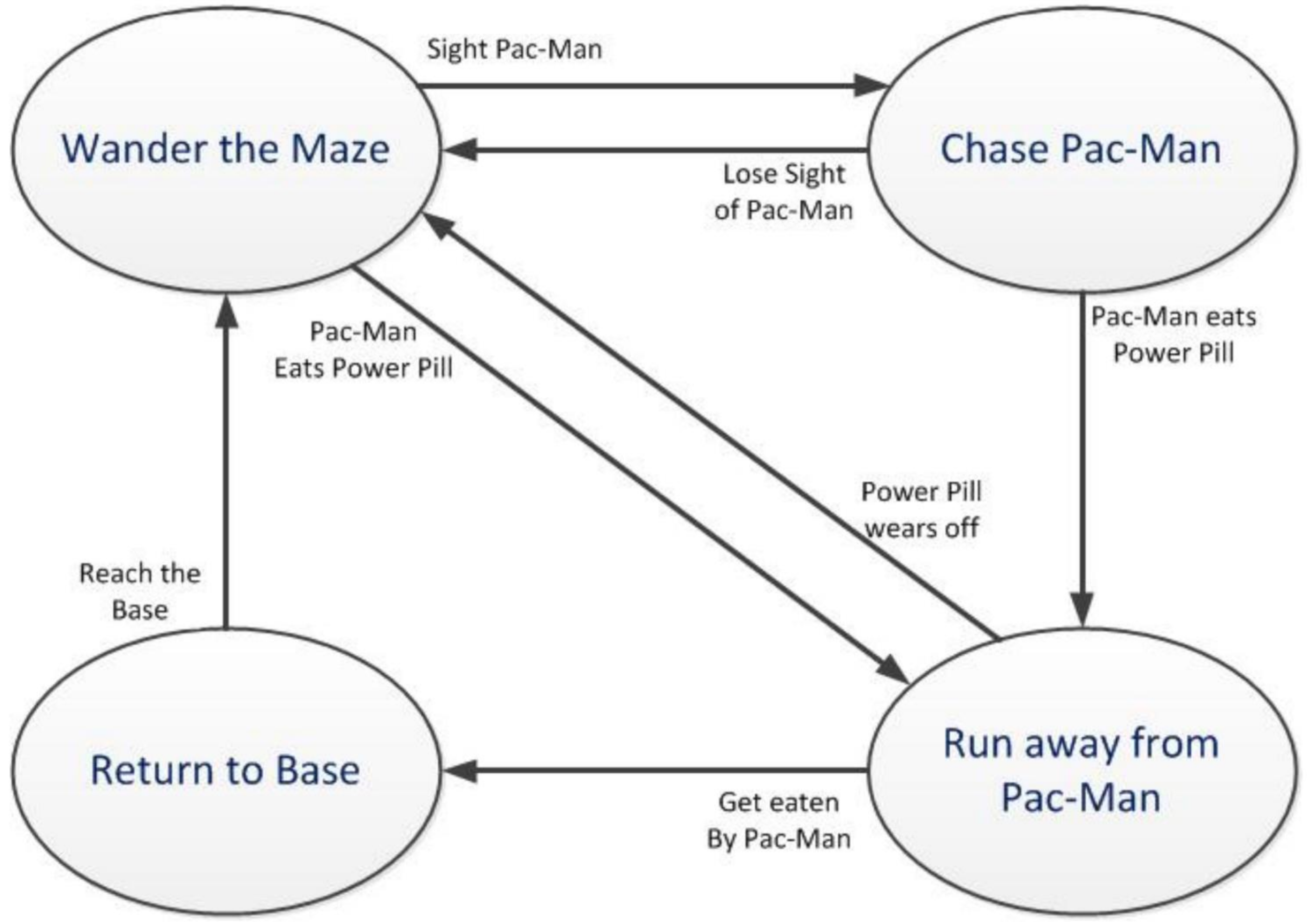
FSM Structure

An FSM can be represented by a graph

- Nodes = states
- Edges = transitions

Each edge has a label informing when the transition should happen

Pacman State Machine

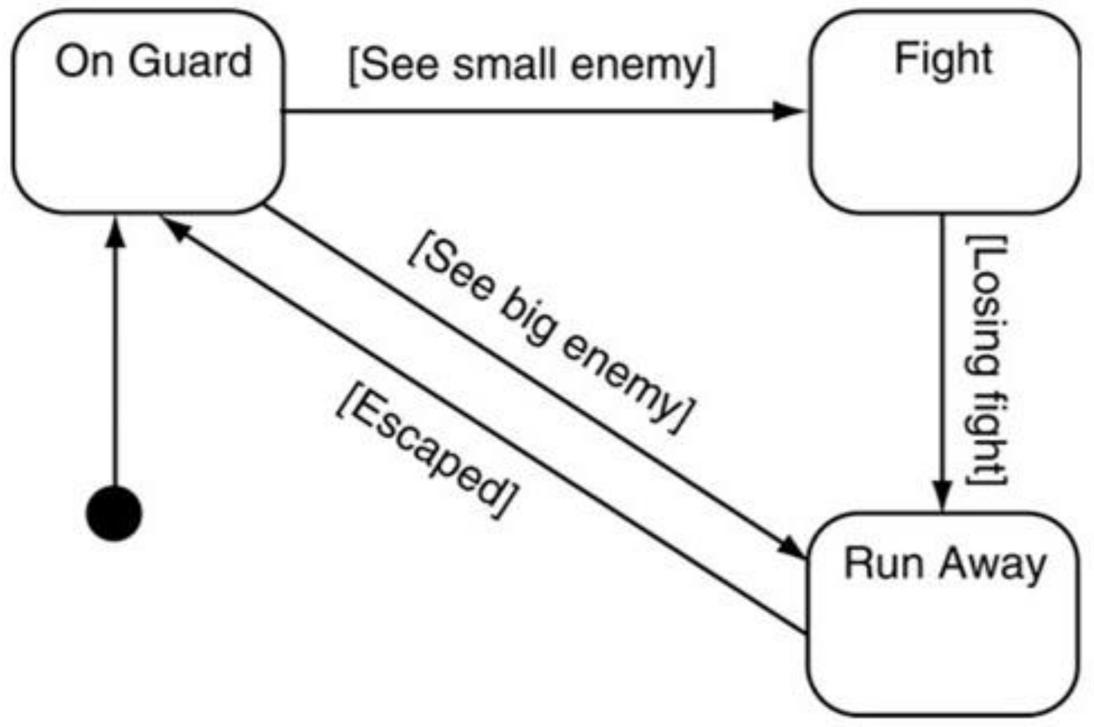


FSM in Games

- Actions or behaviors are associated with each <u>state</u>
- Each <u>transition</u> leads from one state to another, and each has a set of associated conditions
- When the conditions of a transition are met, then the character changes state to the transition’s target state
- Each character is controlled by one state machine and they have a <u>current state</u>
- Often when implemented these are hardcoded

Hard-Coded State Machine

```
enum State { GUARD, FIGHT, RUN };
State myState;

function update() {

  if (myState === GUARD) {
    if seePlayer() { myState = FIGHT; }
    if seeBigEnemy() { myState = RUN; }

  } else if (myState === FIGHT) {
    if losing() { myState = RUN; }

  } else if (myState === RUN) {
    if escaped() { myState = GUARD; }
  }

}
```



Hard-Coded State Machines

Hard-coded state machines are easy to write and fast,
but they can be difficult to maintain

Complex finite states machines can cause spaghetti code

Devs have to write out each behaviour for every individual AI character

The game has to be recompiled each time the behavior changes

Class-Based State Machines

Create an abstract BaseState class with abstract methods:

enterState(), updateState()

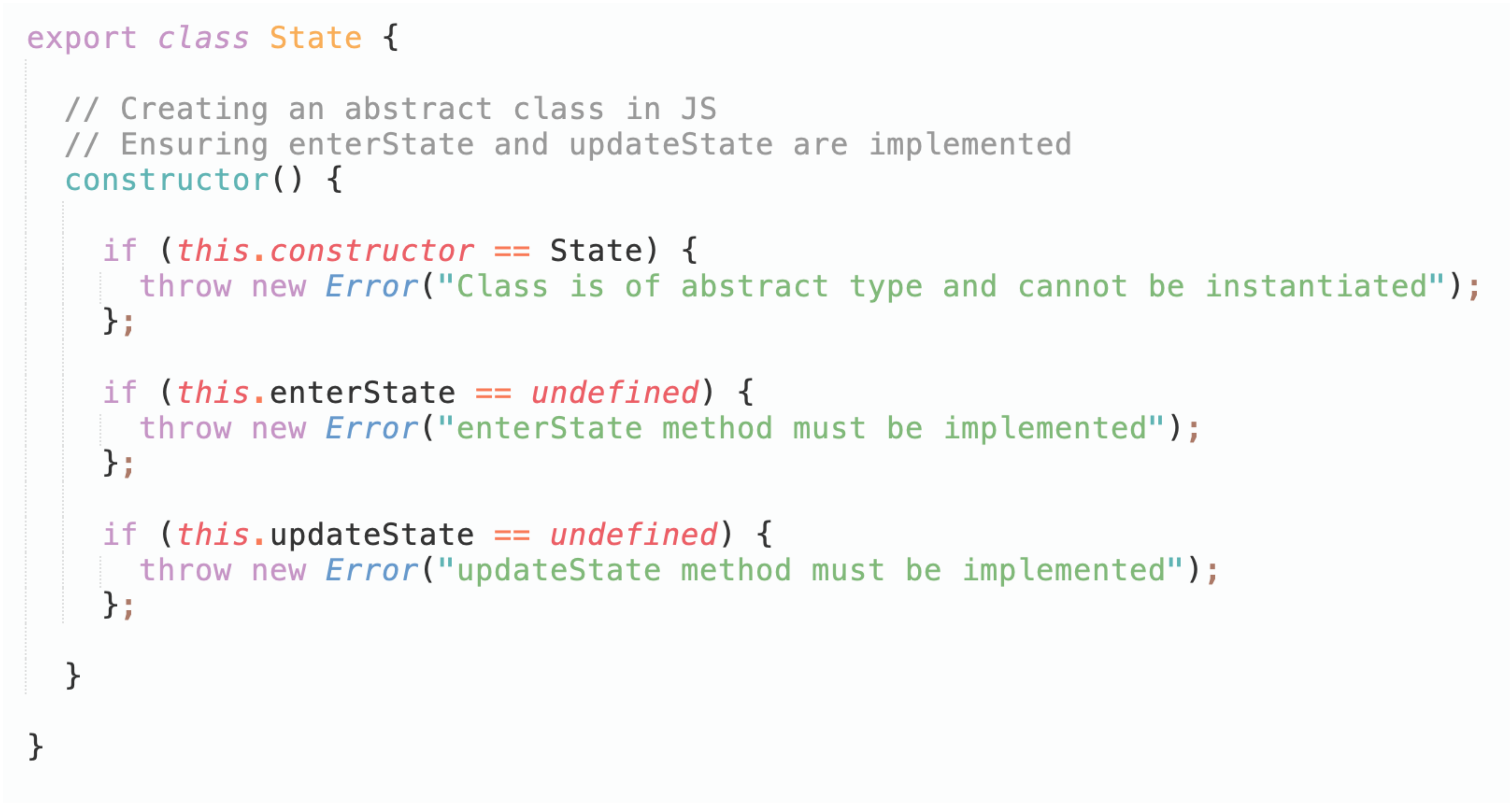
Each state is a <u>separate class</u>:

e.g. class GuardState, class FightState, class RunState

Each state class will implement the generic abstract methods from the BaseState

More flexible/extendable

Abstract Class in JS



```
export class State {

  // Creating an abstract class in JS
  // Ensuring enterState and updateState are implemented
  constructor() {

    if (this.constructor == State) {
      throw new Error("Class is of abstract type and cannot be instantiated");
    };

    if (this.enterState == undefined) {
      throw new Error("enterState method must be implemented");
    };

    if (this.updateState == undefined) {
      throw new Error("updateState method must be implemented");
    };

  }

}
```

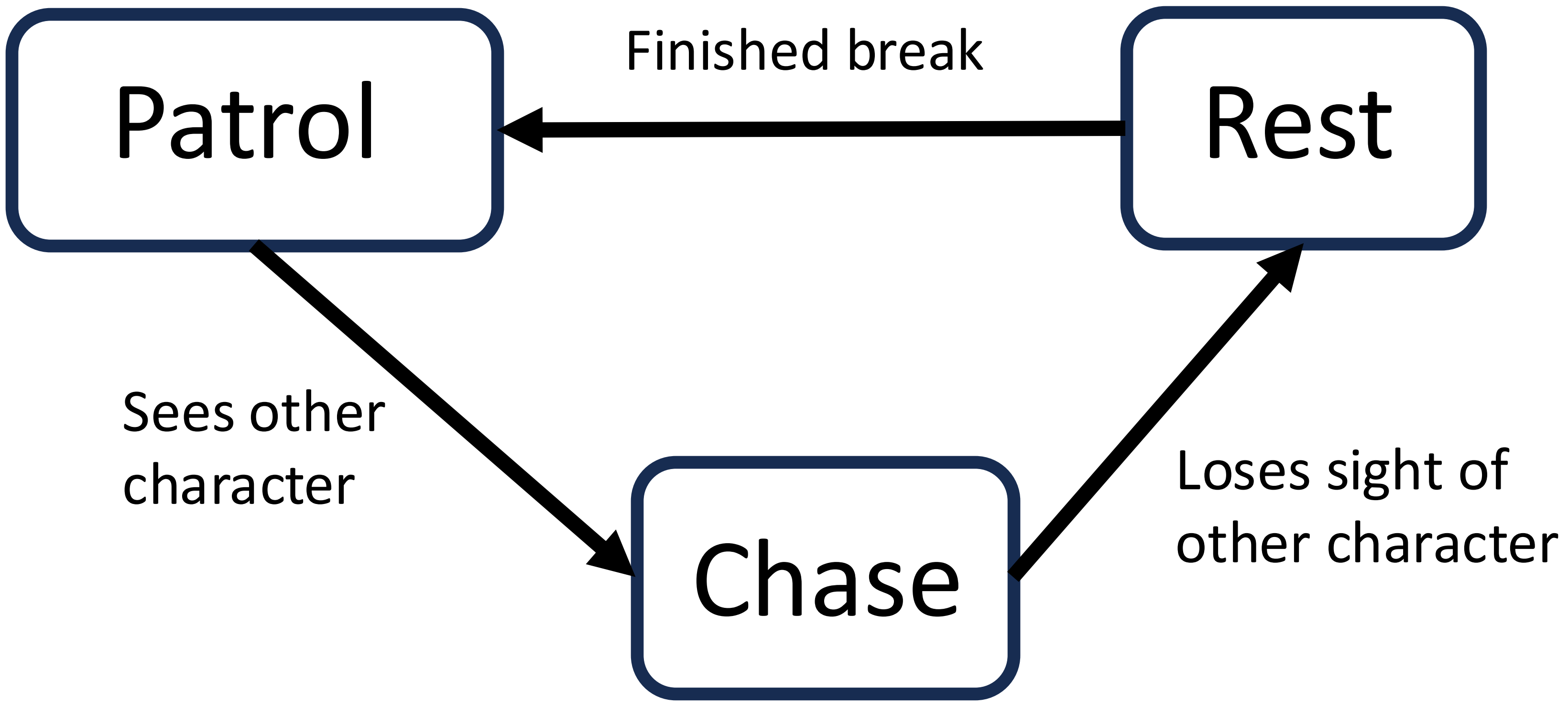
Basic Example

We have a character patrolling the scene

If it sees another character, it speeds up and chases them

If it loses the character, it pauses for a 2 second timeout
(to catch it’s breath) then resumes patrolling

State Machine (Basic Example)



Guard class

Constructor update to take in an enemy

- This is so we can call enterState with an enemy as an argument

Instance variables

- this.state

Methods

- switchState(state, enemy)
  - to call enterState() on our new state
- update(deltaTime, bounds, enemy)
  - call the character's update method and to call updateState() on our state

Notes on main

We will apply steering for our guard when we update our state

So, in our main let's remove:

```
guard.applyForce(guard.wander());
```

And change our update function to include the enemy argument

```
guard.update(deltaTime, bounds, enemy);
```

Patrol State

When patrolling:

Guard will wander at a top speed of 10 (character default is 20, so this is slower than the enemy)

Guard's colour will be set to blue

enterState()

- Set top speed to 10
- Set colour to blue

updateState()

- Check transition: is guard close to our enemy?
    - If so, switch to Fight!
    - Otherwise, wander

Fight State

When fighting:

Guard will seek the enemy at a top speed of 30

Guard's colour will be set to yellow

enterState()

- Set top speed to 30
- Set colour to yellow

updateState()

- Check transition: is guard far away from our enemy?
    - If so, switch to rest
    - Otherwise, seek the enemy

Rest State

When resting:

Guard will slow down to a stop

Guard's colour will be set to white

enterState()

- Set colour to white
- Initialize 2 second timer
    - Once finished, switch to patrol state

updateState()

- apply brakes