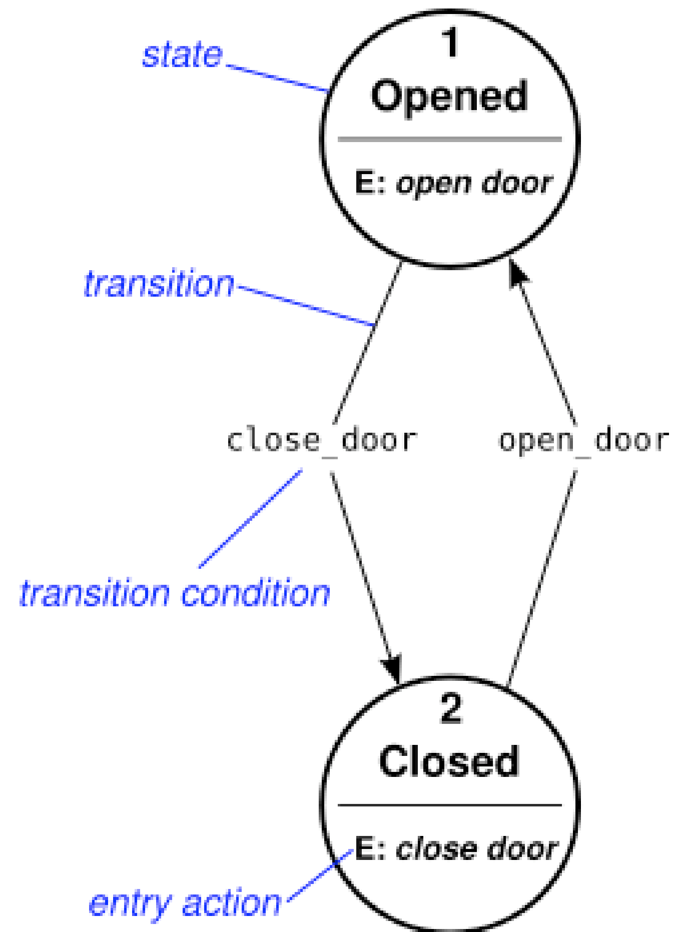
Finite State Machines (FSM)

- FSM is one of the simplest and most basic AI models.
- Basically, FSM consists of
  - States
  - State transitions
- An object (a non-player character) is in one of the states.
- When certain conditions are met, the object changes to another state.

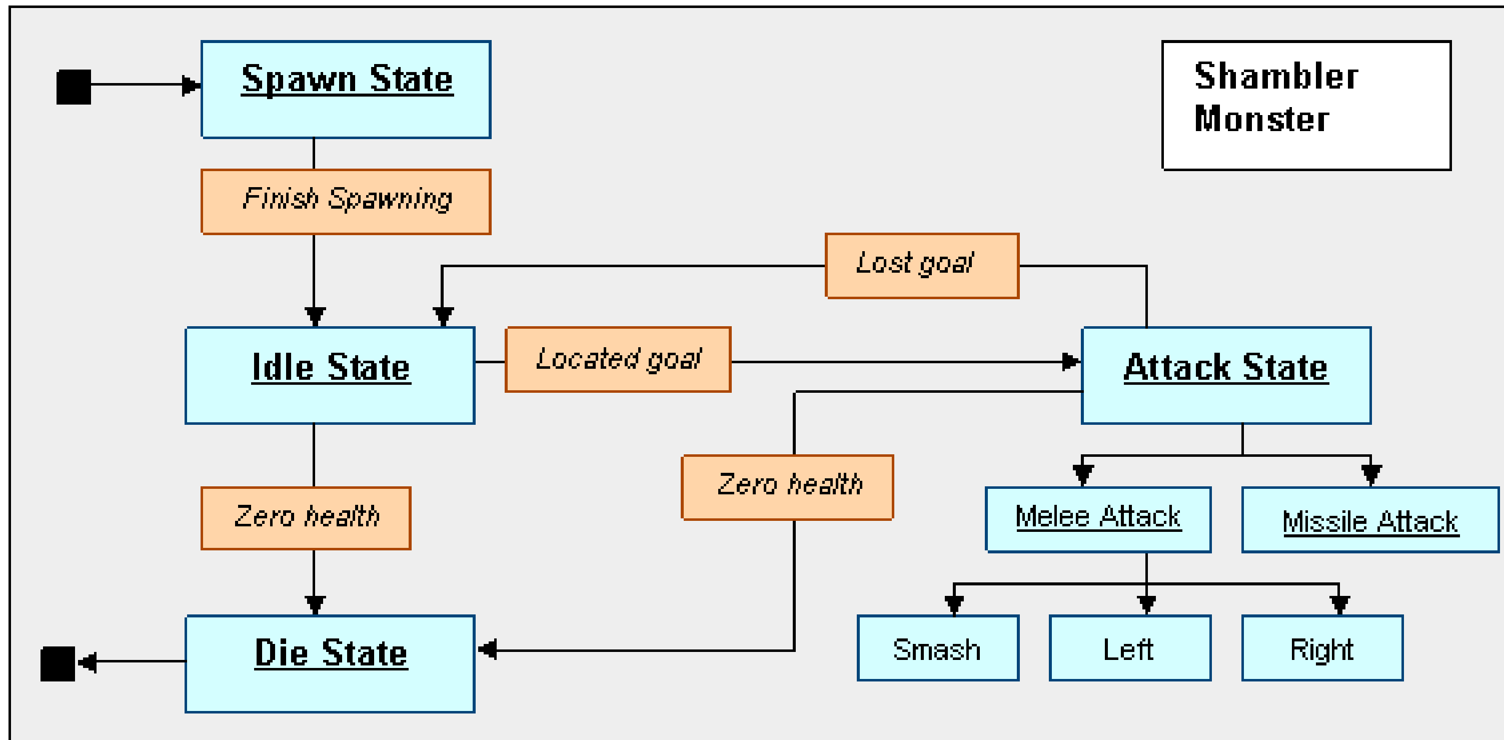
FSM Basics

- A FSM consists of the following 4 components:
  - States which define behavior and may produce actions (Moore Machine)
  - State transitions which are movement from one state to another and may produce actions (Mearly Machine)
  - Rules/conditions/labels which must be met to allow a state transition
  - Input events which are either externally or internally generated, which may possibly trigger rules/satisfy conditions/match labels and lead to state transitions

FSM Basics



FSM Basics



Example of States

- For example, Quake2 uses 9 states for the monsters:
  - Standing, walking, running, dodging, attacking, melee, seeing the enemy, idle and searching
- In other words, at any one time, a monster can be in one of the above 9 states.
- Each state has its own distinct behavior and actions. For example, a monster in the running state would behave differently from a monster in the standing state.
- The behavior and actions for each state has to be defined by the programmer.

State Transitions

- A finite state machine must have:
  - an initial state which provides a starting point, and
  - a current state which remembers the product of the last state transition.
- An input event act as a trigger
- This causes an evaluation of the rules that govern the transitions from the current state to other states. A state change then occurs according to the rules.
- The best way to visualize a FSM is to think of it as a directed graph of the states.

Example of State Transitions

state
state transition
input event

Die
Get Shot
Melee
Get Shot
Get Shot
Close range
Start state
Close range
Idle
See enemy
Attack
Enemy Fires
Dodge
Lose Sight
See enemy
Search
Time Out

Another Example: FSM for Ghost in Pac-Man

Player re-spawns

Move Randomly

Time to exit Center Room

Chase Player

Player dies

Pellet wears off

Player eats Pellet

Player eats Ghost

Run from Player

Rise

Die

Eyes reach Center Room

Disadvantages of FSM

- May be too predictable
- Large FSM with many states and transitions can be difficult to manage and maintain. The graph may start to look like “spaghetti.”
- State oscillation. States may be too rigid. Conditions are too crisp.
  - For example, there are two states: Flee and Idle.
  - The condition for being in the Flee state is to be within a distance 5.0 from the enemy. The condition for being in the Idle state is to be greater than 5.0 from the enemy.
  - Say, the object is 4.9 from the enemy. It is in Flee state, so it runs away. Now it is 5.1, so it is in Idle state. It randomly moves around, and goes to 4.9 and gets into the Flee state again etc.

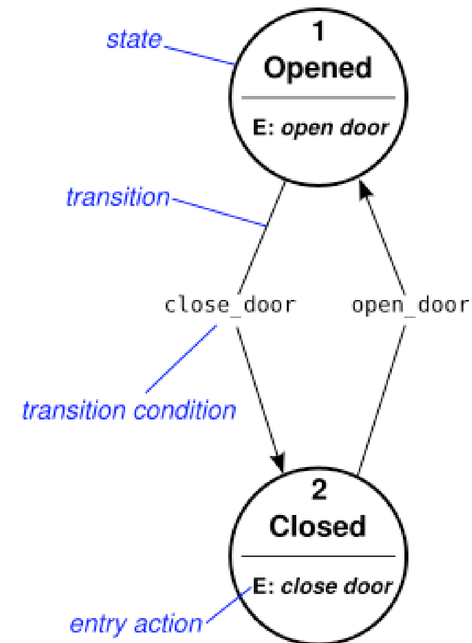
Implementation

```
class Monster {
  int state; // 0: Idle
                 1: Attack
                 2: Melee
                 3: Dodge
                 4: Search
                 5: Die
  Monster();
  void Iterate();
  void HandleInput(int eventID);
  void Shoot();
  void Melee();
  void Dodge();
  void Chase();
  void FindEnemy();
};
```

```
Monster::Monster() : state(0) { } // initialize to start state

void Monster::Iterate() { // called on every game cycle
  if (state==0) { Move(rand()); // move in random direction
  } else if (state==1) { Chase(); Shoot();
  } else if (state==2) { Melee();
  } else if (state==3) { Dodge();
  } else if (state==4) { FindEnemy();
  } else if (state==5) {
  }
}

void Monster::HandleInput(int eventID) {
  if (eventID==0) { // monster got shot
    state = 5;
  } else if (eventID==1) { // see enemy
    if ((state==0) || (state==4)) {
      state = 1;
    }
  } …
}
```

Prolog Implementation

```
% define the machine
initial(closed).
edge(closed,open_door,opened).
edge(opened,close_door,closed).

% define the interpretation of the machine
behavior :-
    initial(State),
    edge(State,Action,NextState),
    writeln(Action),
    transition(NextState).

transition(State) :-
    edge(State,Action,NextState),
    writeln(Action),
    transition(NextState).
```



Prolog Implementation

```
% define the machine
initial(search).
edge(search,tofu_found,found).
edge(found,tofu_pickup,picked_up).
edge(picked_up,tofu_retrieve,retrieved).
final(retrieved).

% define the interpretation of the machine
behavior :-
    initial(State),
    edge(State,Action,NextState),
    writeln(Action),
    transition(NextState).

transition(State) :-
    final(State).

transition(State) :-
    edge(State,Action,NextState),
    writeln(Action),
    transition(NextState).
```

What does the machine look like?