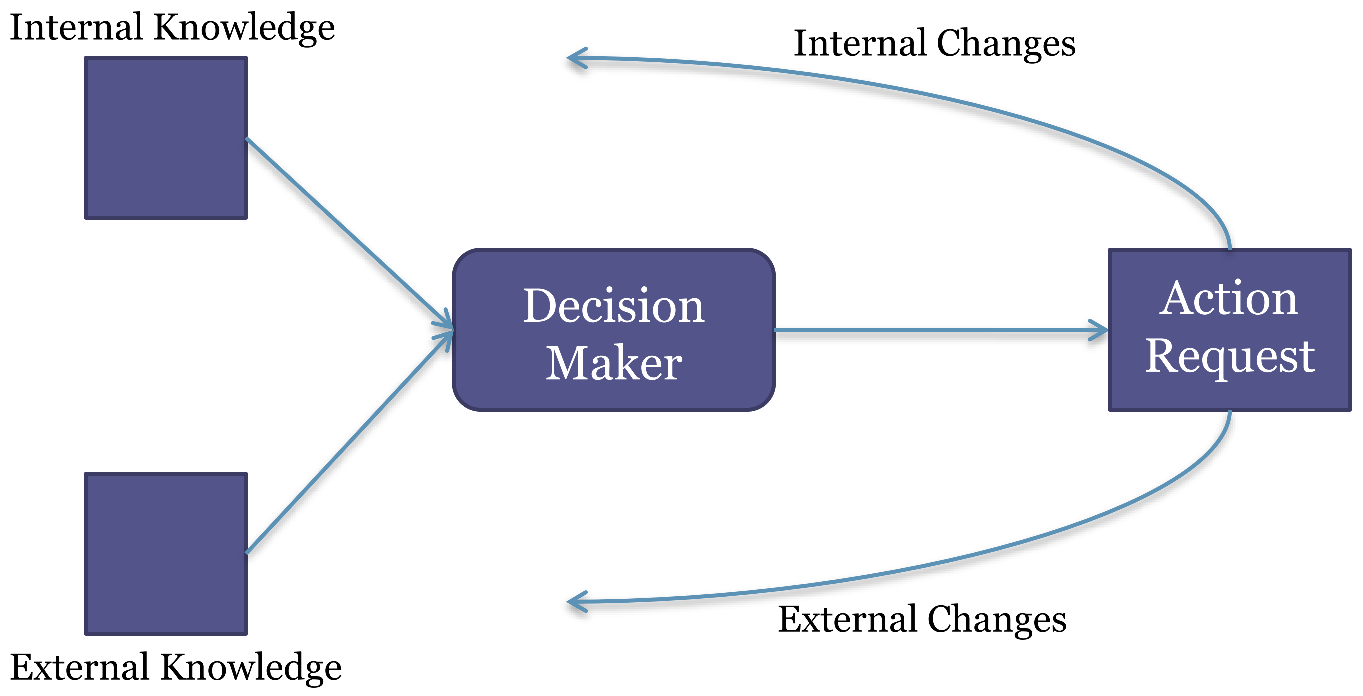
CSD1130
Game Implementation Techniques

Lecture 15

Outline

- Decision Making
  - Decision Trees
  - State Machines
    - FSM (Finite State Machines)
    - Hierarchical State Machines
- FSM Scripting Language
- AI Behavior in Platformer

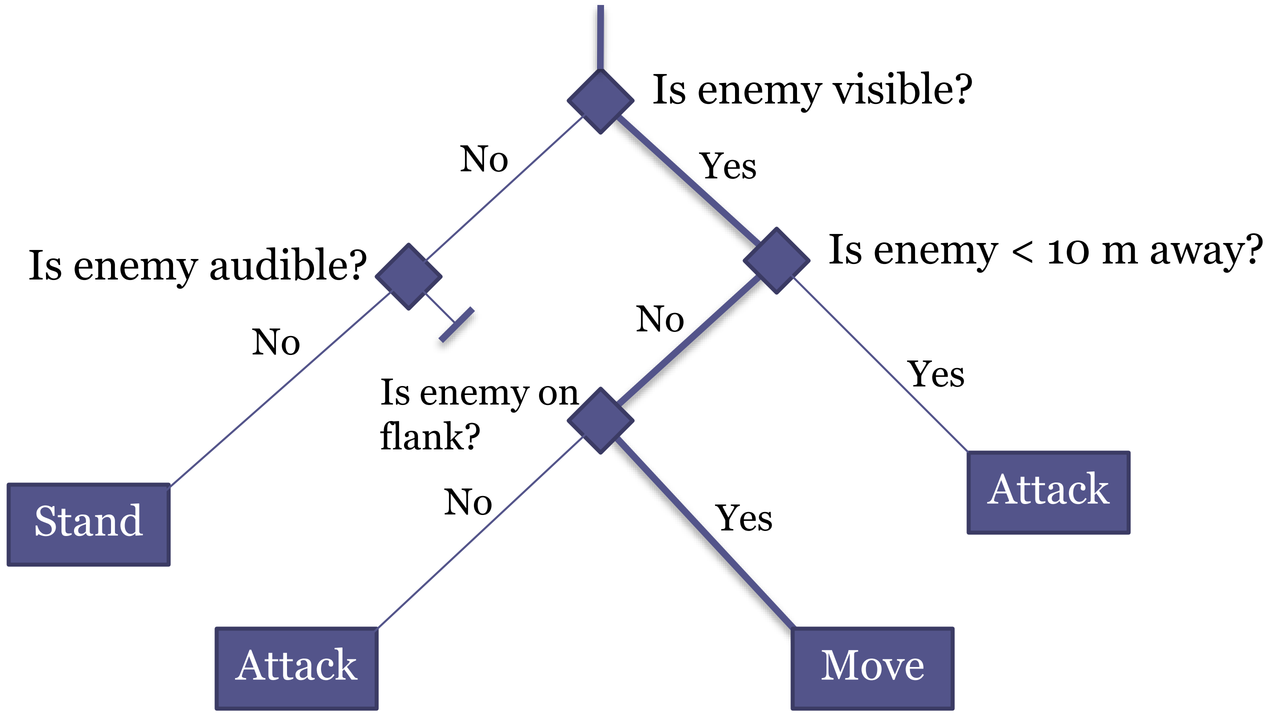
Decision Making (1/2)



Decision Making (2/2)

- The input is the knowledge that a character possesses. The knowledge could be broken down into two sections:
  - External knowledge
  - Internal Knowledge
- The output would be the action

Decision Trees (1/2)



Decision Trees (2/2)

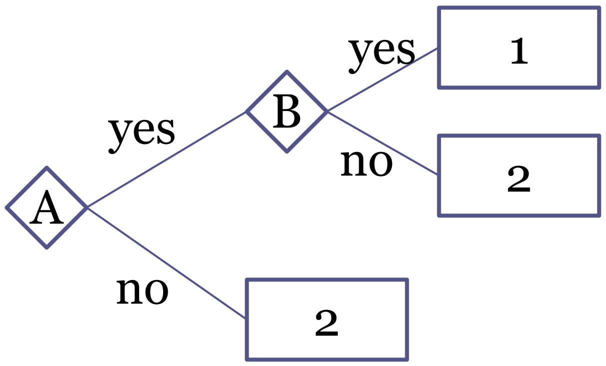
- A decision tree is just a series of questions that help a unit decide what to do given its current situation
    - Often a series of yes/no questions
    - This leads to a *binary tree*
- Each terminal (or *leaf*) *node represents a state*, e.g., *attack, hide, patrol, etc.*

Decision Trees Node

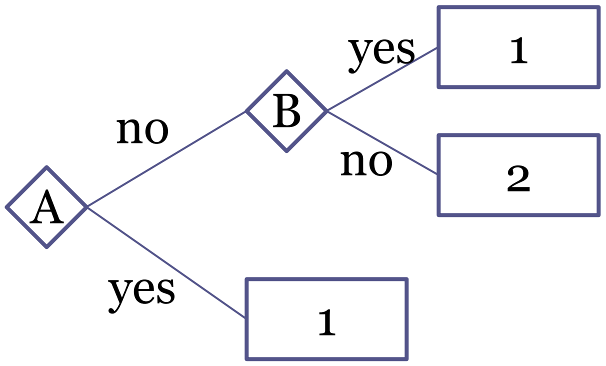
```
struct NodeTree
{
        //void * data; //or int //or object *...
        NodeTree * left;
        NodeTree * right;

        virtual NodeTree * Execute() = 0;
        void MakeDecision();
};
```

Combinations of Decisions

If A AND B then action 1, otherwise action 2

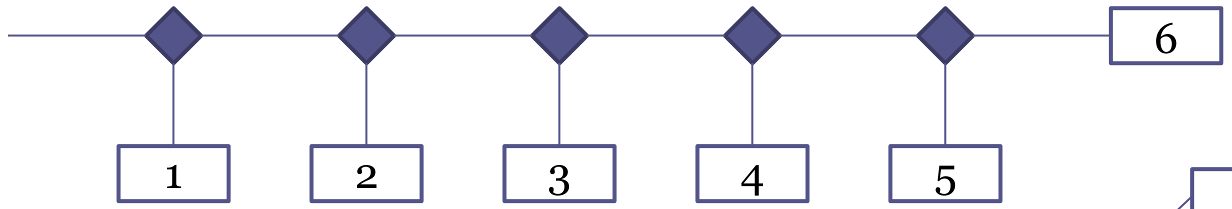


If A OR B then action 1, otherwise action 2

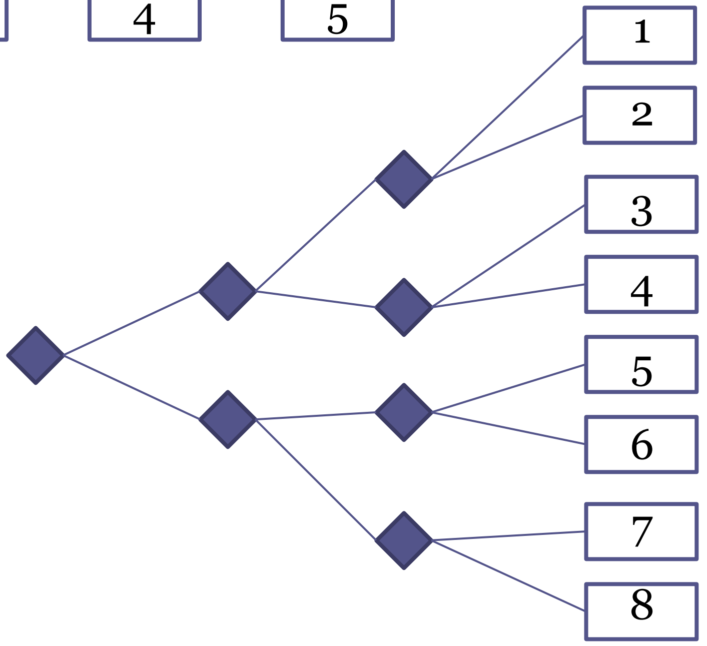


Balancing Trees

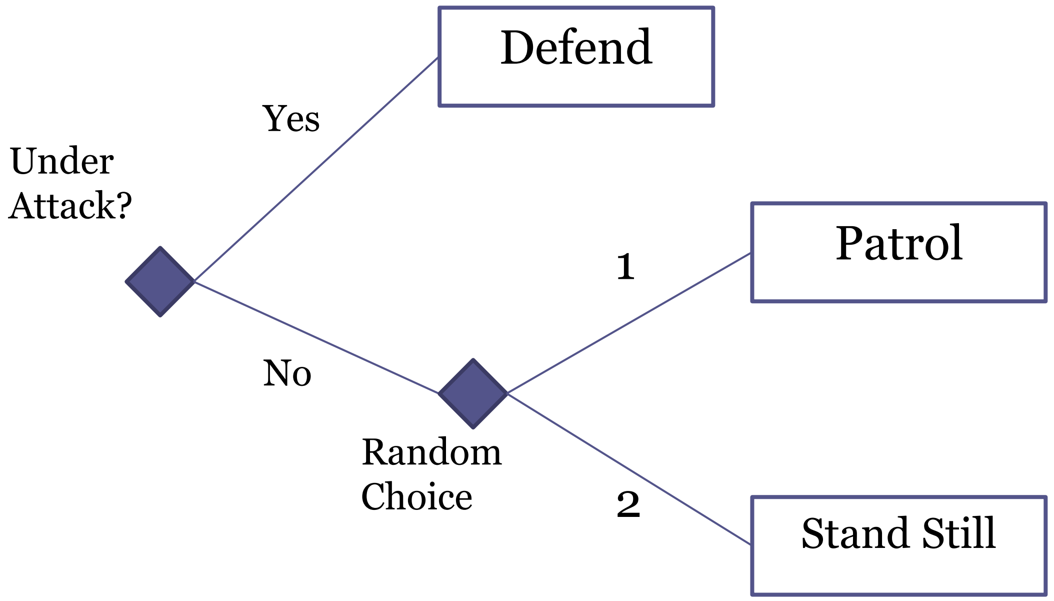
Unbalanced tree



Balanced tree



Random Decision Trees (1/2)



Random Decision Trees (2/2)

```
struct RandomDecision (Decision)
   lastFrame = -1
   lastDecision = false

def  test()
   # check if our stored decision is too old
   if frame() > lastFrame + 1
      # make a new decision and store it
      lastDecision = randomBoolean()

   # either way we need to update the frame value
   lastFrame = frame()

   # we return the stored value
   return lastDecision
```

Outline

- Decision Making
  - Decision Trees
  - State Machines
    - FSM (Finite State Machines)
    - Hierarchical State Machines
- FSM Scripting Language
- AI Behavior in Platformer

Finite State Machines

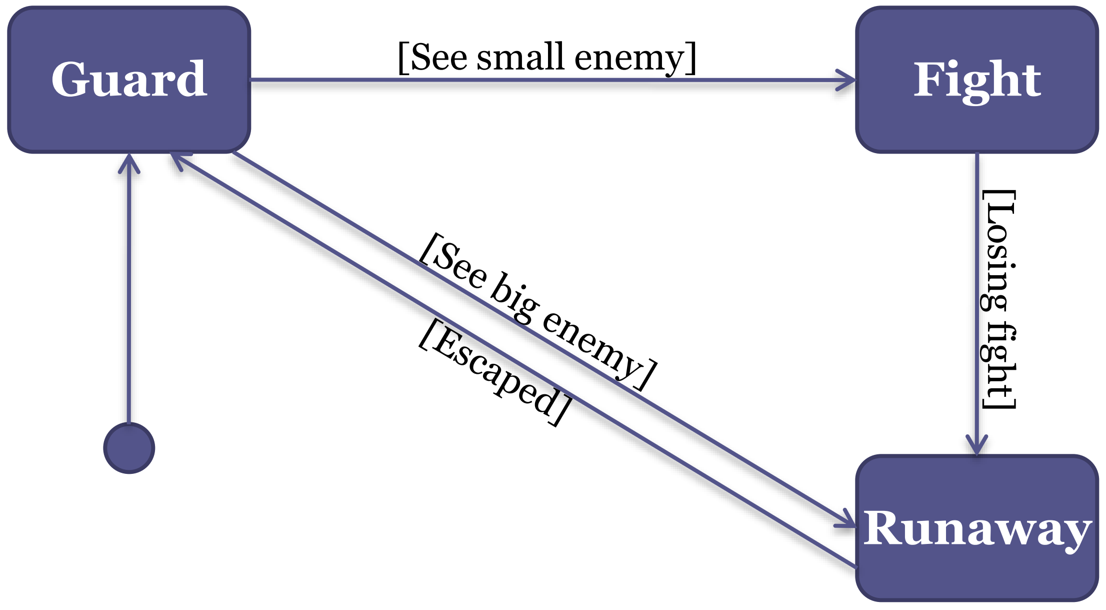


*Diagram based on UML state chart diagram format

A Generic Implementation (1/3)

- The state machine manager keeps track of the set of possible states and records the current state it is in.

- The transition simply reports to the state machine manager whether its condition is triggered or not.

A Generic Implementation (2/3)

- At each iteration, the state machine manager's update function is called and checks if any transition from the current state occurred.

- Transitions can have priorities and the higher in priority is called.

A Generic Implementation (3/3)

- It is important not to assign a high priority for a transition, where its condition is the condition's subset of a transition with a lower priority.

- Ex:
  Transition 1: if (left)
  Transition 2: if (left and up)

Pseudo-Code

```
class State:
        m_transitions //Holds the list of transitions

        getAction ()
        getEntryAction ()
        getExitAction ()
         getTransitions ()
```

```
class Transition:
        isTriggered ()
        getTargetState ()
        getCondition ()
```

```
class FSM:
        m_states  //Holds the list of states
        m_initialState
        m_currentState

        Update()  //Checks and applies transitions
```

Hard-Coded FSM

Example:

```
class FSM:
        enum State: PATROL, DEFEND, SLEEP
        myState   //Holds the current state

        Update()
                if  myState == PATROL:
                        # example transitions
                        if  canSeePlayer(): myState = DEFEND
                        if  tired(): myState = SLEEP
                elif  myState == DEFEND:
                        if  not  canSeePlayer(): myState = PATROL

                elif  myState == SLEEP:
                        if  not  tired (): myState = PATROL
```

Pros and Cons

- Easy to write but difficult to maintain
- Fast to implement for all but huge state machines
- The need to write the AI behavior for each character
- Hierarchical state machines are difficult to coordinate using hard-coded FSM

FSM with Macros (1/3)

Example:

```
bool  FSM::States ( StateMachineEvent  event,  int  state)
{
   BeginStateMachine
                State ( STATE_0 )
                        OnUpdate
                                Wander ();
                                if ( SeeEnemy () ) SetState (STATE_1 );
                                if ( Dead () ) SetState (STATE_2 );
                State (STATE_1 )
                        OnUpdate
                                Attack ();
                                SetState (STATE_0 );
                                if ( Dead () ) SetState (STATE_2 );
                State (STATE_2 )
                        OnUpdate
                                RotSlowly ();
  EndStateMachine
}
```

FSM with Macros (2/3)

Example:

```
bool   FSM::States ( StateMachineEvent  event,  int  state)
{
   if( state < 0 ) {
        if( 0 ) {
            return ( true );
        }
    } else if( state == STATE_0) {
                if( 0 ) {
                                   } OnUpdate
                                                  Wander ();
                                                  if ( SeeEnemy () ) SetState (STATE_1 );
                                                  if ( Dead () ) SetState (STATE_2 ); }
                  State (STATE_1 )
                                   OnUpdate
                                                  Attack ();
                                                  SetState (STATE_0 );
                                                  if ( Dead () ) SetState (STATE_2 );
                  State (STATE_2 )
                                   OnUpdate
                                                  RotSlowly ();
  EndStateMachine
}
```

FSM with Macros (3/3)

- Pros:
  - Structure
    - All state machines have a consistent format
  - Readability
  - Debugging

- Cons
  - Hard to maintain (when adding macros) so it is important to determine requirements at the beginning

Macros are like Legos

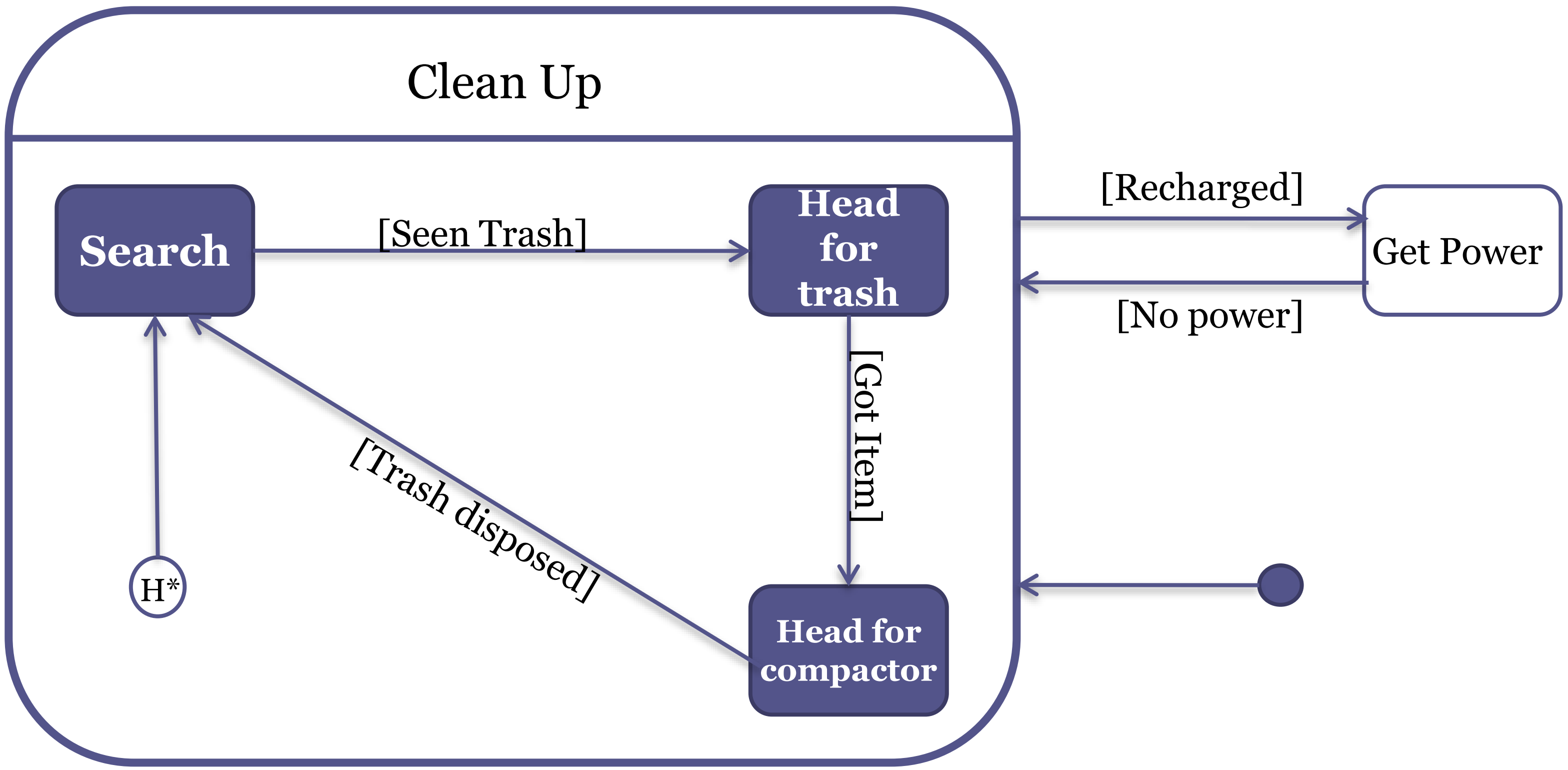
```
bool Agent::States( StateMachineEvent event,
                    MSG_Object * msg, int state )
{    if( state < 0 ) {
         if(0) {
             return( true );
         }
     } else if( STATE_Name == state ) {
         if(0) {
             return( true );
         } else if( EVENT_Enter == event ) {
             //C++ code 1
             return( true );
         } else if( EVENT_Message==event && MSG_Name==msg->GetMsgName() ) {
             //C++ code 2
             return( true );
         }
     } else {
         assert( 0 && "Invalid State" );
         return( false );
     }
     return( false );
}
```

```
BeginStateMachine
    DeclareState(STATE_Name)
        OnEnter
            //C++ code 1
        OnMsg(MSG_Name)
            //C++ code 2
EndStateMachine
```

Outline

- Decision Making
  - Decision Trees
  - State Machines
    - FSM (Finite State Machines)
    - Hierarchical State Machines
- FSM Scripting Language
- AI Behavior in Platformer

Hierarchical State Machines



Outline

- Decision Making
  - Decision Trees
  - State Machines
    - FSM (Finite State Machines)
    - Hierarchical State Machines
- FSM Scripting Language
- AI Behavior in Platformer

FSM Scripting Language (1/2)

- Used by designers
- No compilation needed when changes are done

FSM Scripting Language (2/2)

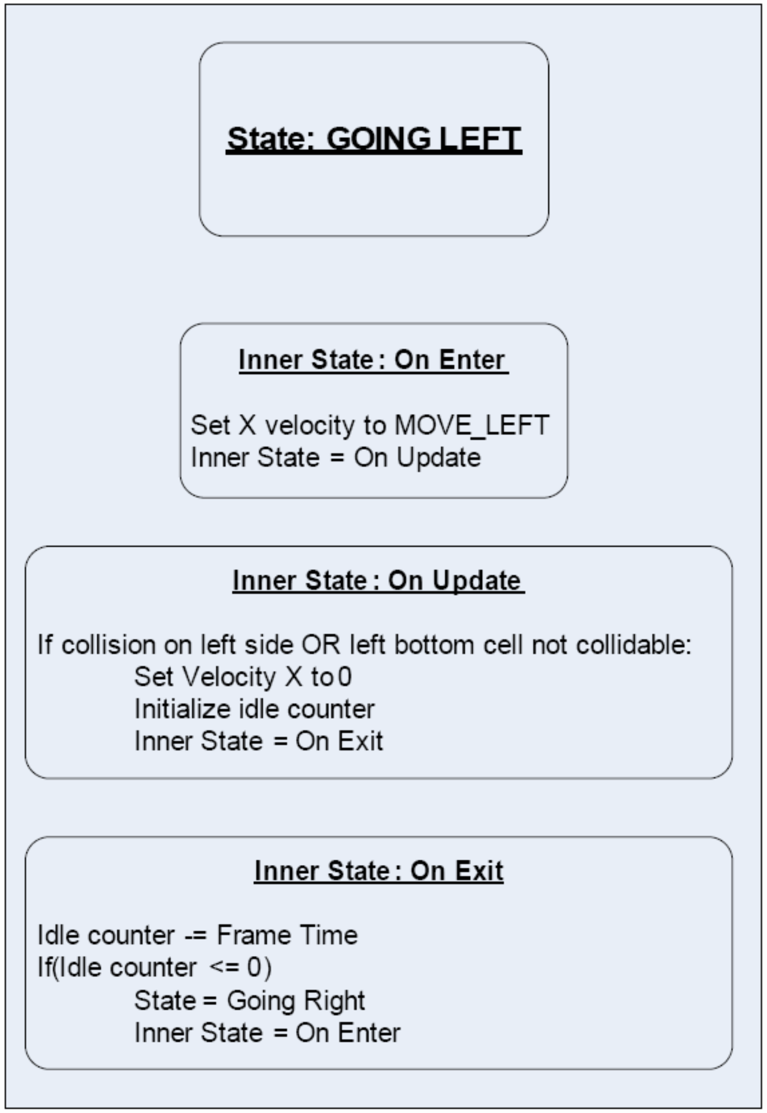
Example:

```
Behavior Patrol
begin

variable
        integer  targetActor

transition
        # listen for enemies
        if  ( targetActor )
             switch ( “Attack” , targetActor )

sequence
        # Patrol
        do forever
        begin
            goto  GetLocation ( “WaypointA” )  walk
            idle 2
            # do more stuff
        end
end
```

Outline

- Decision Making
  - Decision Trees
  - State Machines
    - FSM (Finite State Machines)
    - Hierarchical State Machines
- FSM Scripting Language
- AI Behavior in Platformer

Platformer AI



State: GOING LEFT

Inner State : On Enter

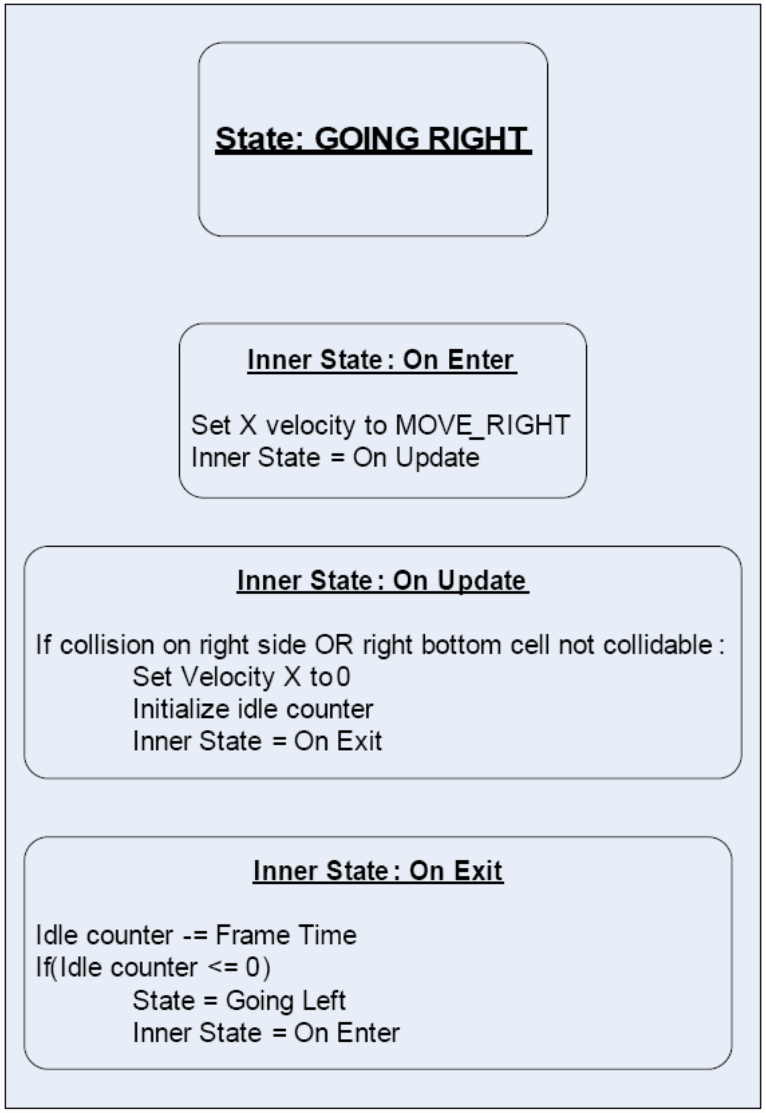
```
Set X velocity to MOVE_LEFT
Inner State = On Update
```

Inner State : On Update

```
If collision on left side OR left bottom cell not collidable:
        Set Velocity X to 0
        Initialize idle counter
        Inner State = On Exit
```

Inner State : On Exit

```
Idle counter -= Frame Time
If(Idle counter <= 0)
        State = Going Right
        Inner State = On Enter
```



State: GOING RIGHT

Inner State : On Enter

```
Set X velocity to MOVE_RIGHT
Inner State = On Update
```

Inner State : On Update

```
If collision on right side OR right bottom cell not collidable :
        Set Velocity X to 0
        Initialize idle counter
        Inner State = On Exit
```

Inner State : On Exit

```
Idle counter -= Frame Time
If(Idle counter <= 0)
        State = Going Left
        Inner State = On Enter
```

References

- Artificial Intelligence for Games by Ian Millington
- AI Game Programming Wisdom 2 by Steve Rabin