Finite State Machines

Defining A.I. with state graphs and transitions

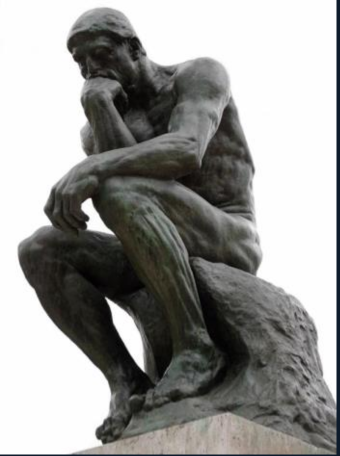
Lecture Contents

- Decision Making
- What is a Finite State Machine?
- States
- Transitions
- Implementation – Switch statement
- More advanced ideas
  - Global states and Transitions
  - Persistent Information
- Implementation – State Machine

Decisions

- Decision making is perhaps the biggest aspect of AI in games
  - Steering and Pathfinding control the motions, but decisions control when/where/how to perform those motions

- There are many tools available to programmers for making decisions
  - Finite State Machines
  - Decision Trees
  - Behaviour Trees
  - Utility Systems
  - Planners
  - Blackboards
  - Fuzzy Logic
  - Neural Networks and Genetic Algorithms
  - Many more!



Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN
GAMES, ANIMATION & FILM VFX

What is a Finite State Machine?

- A finite state machine is a way to split up what an agent can do into discrete chunks that a computer can understand.

- It is a graph, where each node is a specific behaviour for the agent to perform.
  - Chase player
  - Follow patrol path
  - Take cover
  - Run away
- The edges are the transitions between states governed by conditions
  - Can I see the player?
  - Do I have more than 50 health?
  - Are my shields empty?
  - Do I have any Ammo?

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

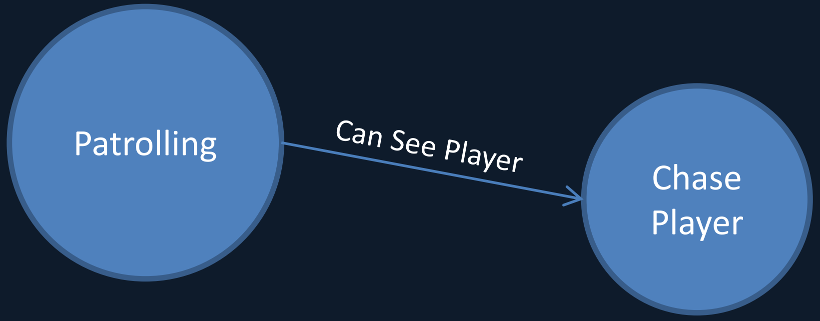
Lets look at a simple example

- We have a simple agent for a stealth game.

- A general description of what we want it to do might look like:
  - Walk along its current patrol
  - If it can see the player – chase the player
  - If its chasing the player and can't see the player – search for the player
  - If it can't find the player in 15 seconds, go back to patrolling
  - If it finds the player while searching start chasing it again
  - If its chasing the player and its less that 5 units away, attack the player
  - If attacking and the player is more than 5 units away, go back to chasing

Lets look at a simple example

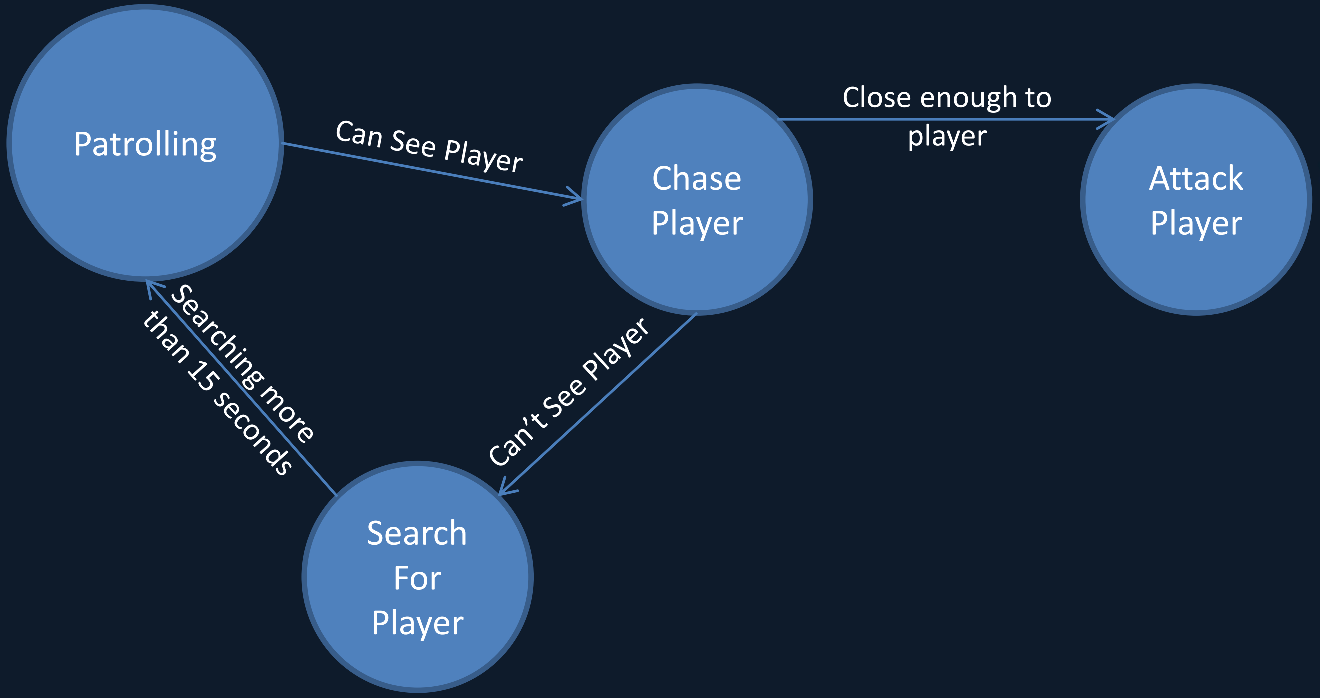
Lets look at a simple example



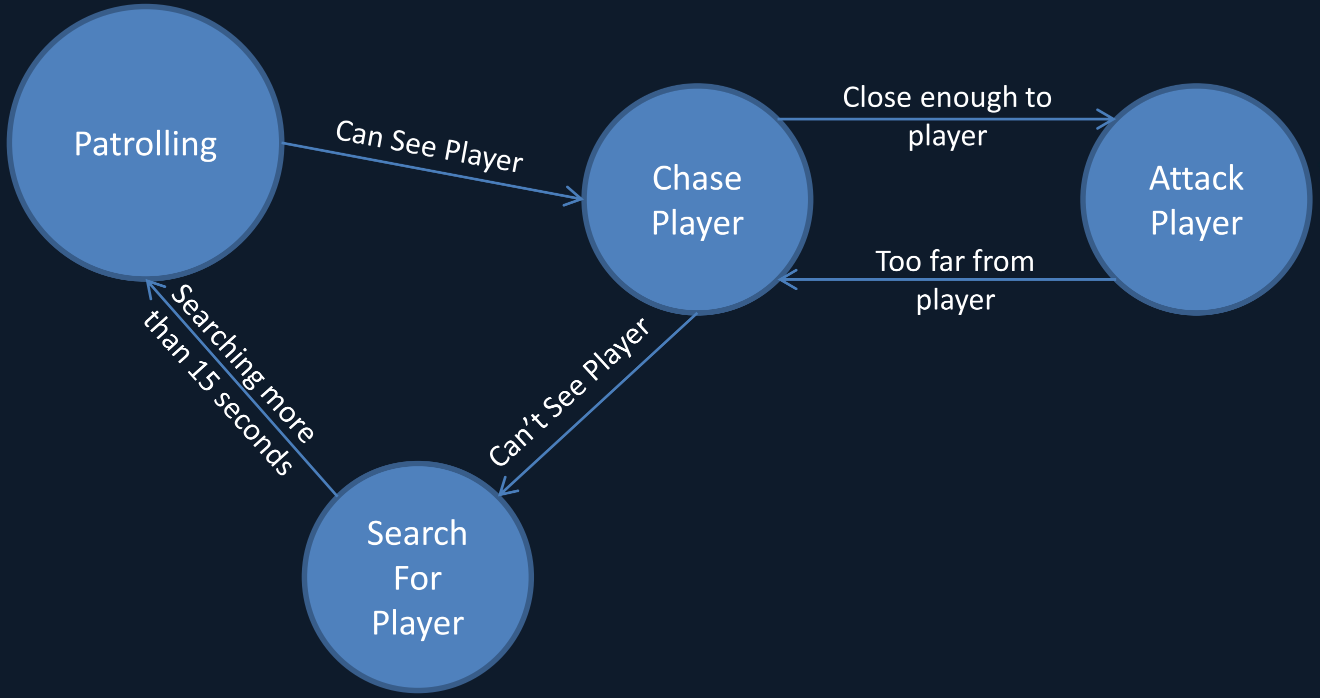
Lets look at a simple example

Lets look at a simple example

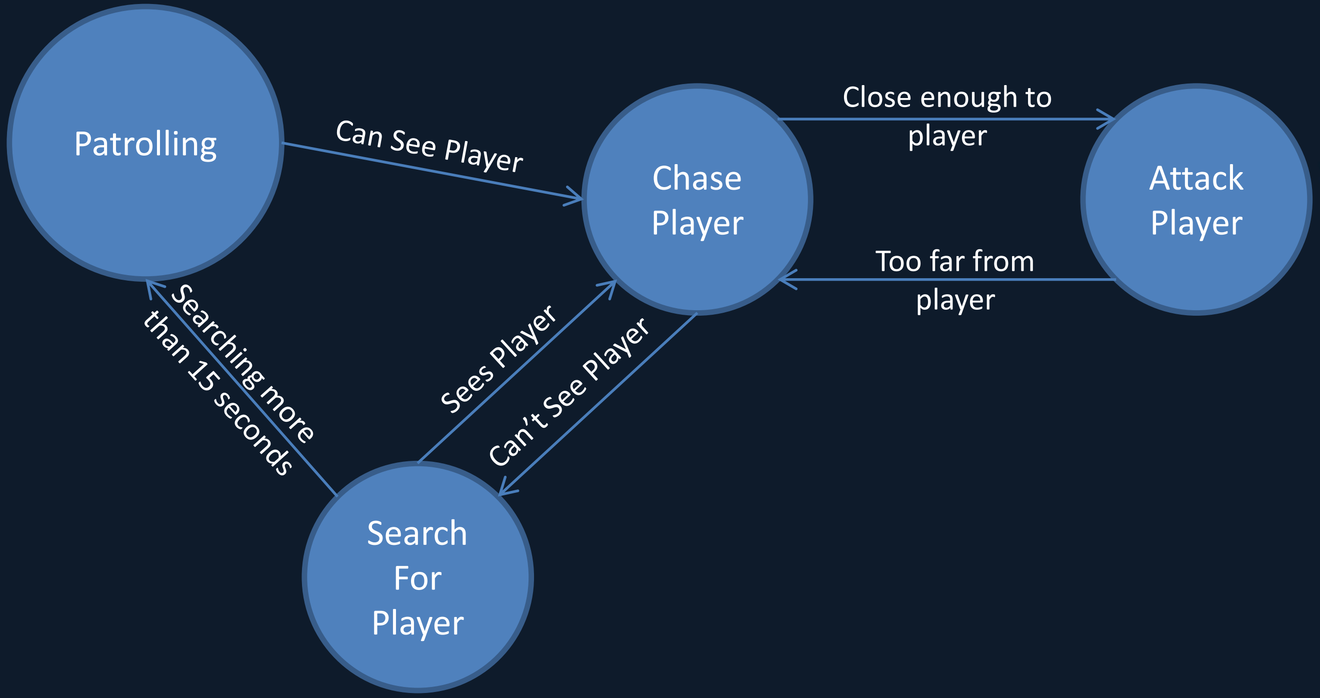
Lets look at a simple example



Lets look at a simple example



Lets look at a simple example



Implementation – Switch Statement

- This is all well and good, but how do we actually implement this?

- The simplest way is just with a switch statement!

Implementation – Switch Statement

- We first create an enum with all the different states our agent can be in.
- Each agent has an instance of the enum that has its current state.
- In its update function, we just have a switch statement that calls a different function depending on what the current state is.
- In each state function we just run the commands for that state and then check each condition to see if we need to make a transition.
  - If we do, we just change the value of the enum

Implementation – Switch Statement

```
void AI_Update( AI* a_poAI )
{
  switch( a_poAI->state )
  {
    case PATROLLING: //AI follows its pre-defined patrol
      AI_Patrol( a_poAI );
    break;
    case CHASE:    //Chase the player
       AI_Chase( a_poAI );
    break;
    case ATTACK:   //Attack the player
       AI_Attack( a_poAI );
    break;
    case SEARCH:   //Search for the player
       AI_Search( a_poAI );
    break;
   }
}
```

Implementation – Switch Statement

- You've already used this kind of FSM in your assessments for handling your game-states.

- Advantages:
  - Easy to implement and understand.
- Disadvantages
  - Doesn't scale well

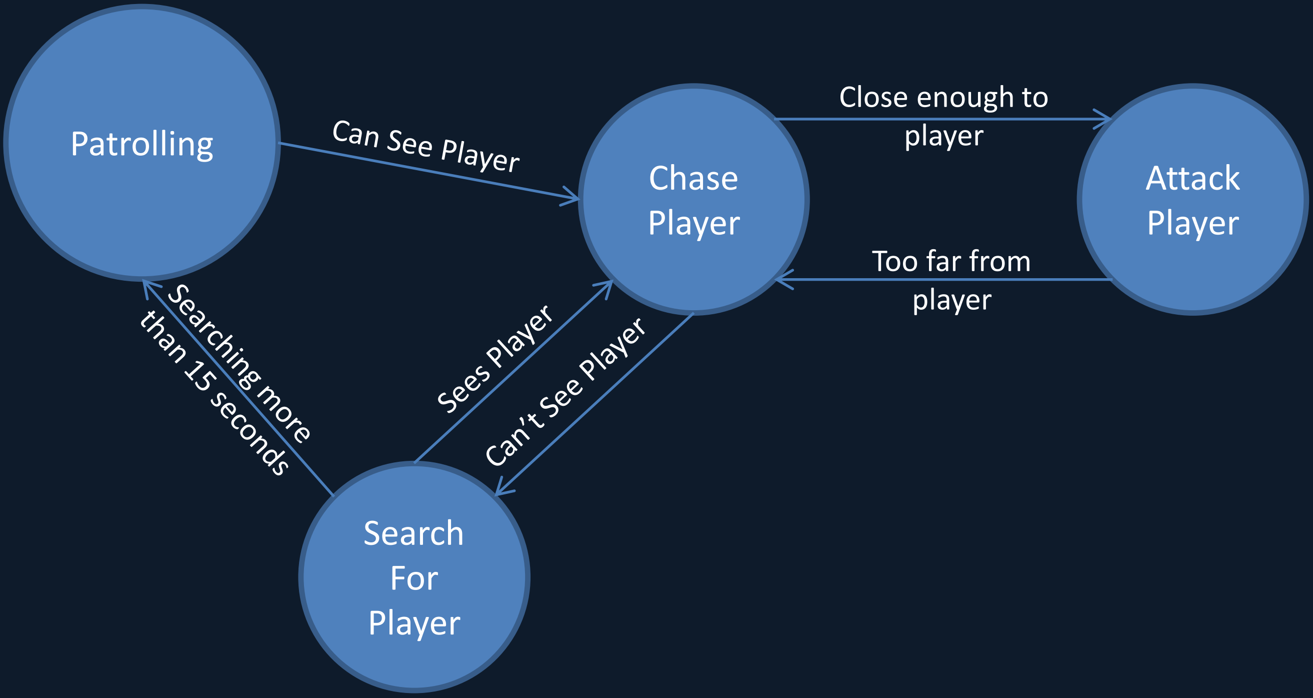
Doing more with State Machines

- Lets look back at our enemy from before.

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Doing more with State Machines

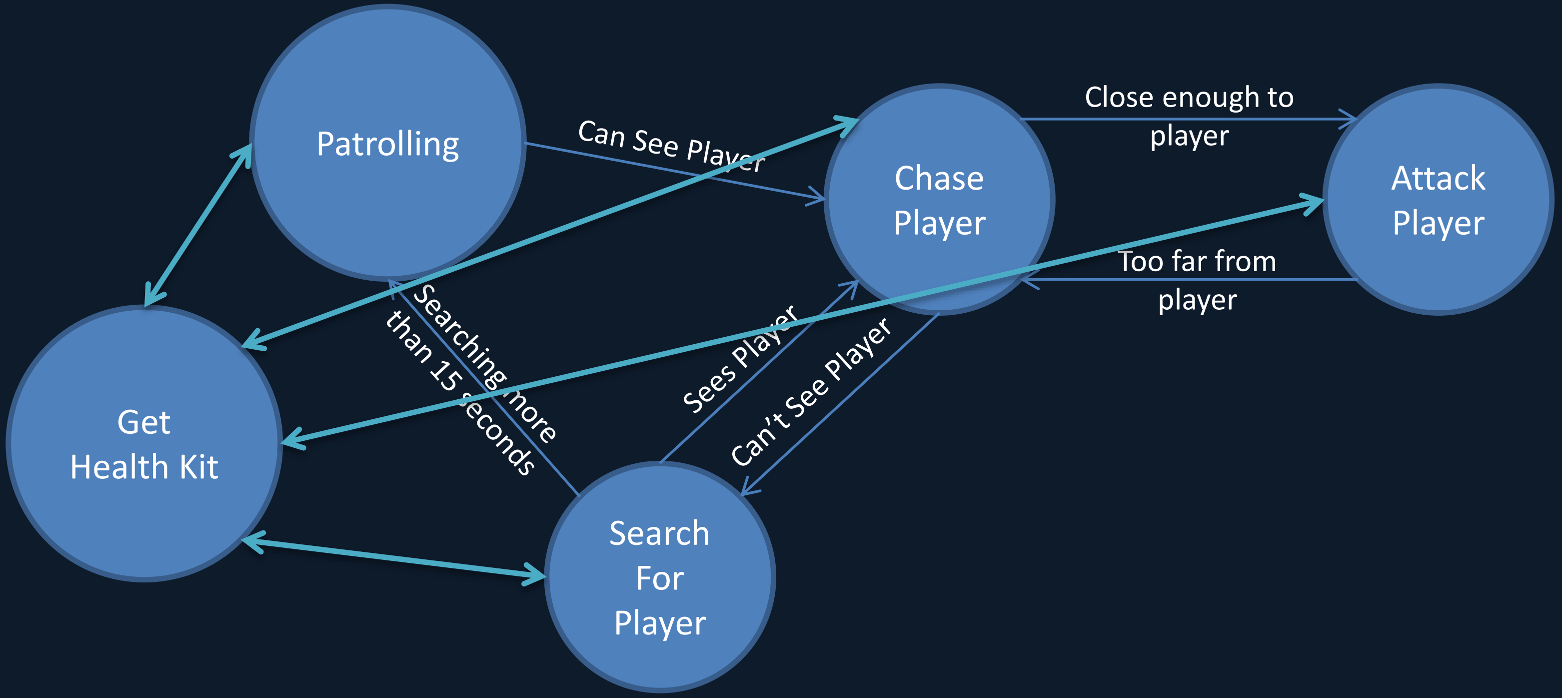


Doing more with State Machines

- Say we wanted our agent to go find and pick up a health pack if he goes below a certain health level, regardless of existing state.

- Once he's picked up a health pack, we want him to go back to what he was doing.

Doing more with State Machines



Doing more with State Machines

- The problems with this should be immediately apparent.
  - We’re going to add a lot transitions. It could be really easy to forget to add a transition to the GetHealthKit state.
  - How do we know what state we came from to transition back?

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Doing more with State Machines

- The solutions to these problems are actually quite simple.
  - Have some transitions global, so they are checked regardless of the current state
  - Keep track of not only the current state, but also the previous state.

Doing more with State Machines

- We could just add these to our switch statement system, but it would start to get very messy very quickly.
- We need a better implementation!

Canberra Institute of Technology

aie

SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Implementation – State Machine

- When we made a graph class, we made classes for the GraphNodes, and the Graph itself.

- Lets do something similar for our FSM.

State Class

- Our state class is an abstract base class that represents any state.
- The data for a state is the instructions for the AI to perform.
- So instead of having variables, our state class will have 3 pure virtual functions
  - Init
  - Update
  - Exit
- This gives us more freedom than the switch. We now have self contained functions for what to do when we enter, exit and update a state

```
class State
   func Update(Agent agent) = 0
   func Init(Agent agent) = 0
   func Exit(Agent agent) = 0
```

StateMachine class

- The state machine class is how we manage our current state and traverse the graph.

- The state machine needs to hold two State pointers
  - One for the current state
  - One for the previous state

- Now each of our agents would have an instance of the state machine inside them.
  - We have the agent pass itself into the Update function, which in turn passes it into the state update function.
  - The state needs the agent so it can tell the agent what to do

```
class StateMachine
    func Update(Agent agent)
    func ChangeState(Agent agent, State newState)
    func GetCurrentState()
    func GetPrevState()

    State currentState
    State prevState
```

State Class

- Looking back at our state class, remember we want our State functions to have control over changing to different states.

```
class State
    func Update(Agent agent) = 0
    func Init(Agent agent) = 0
    func Exit(Agent agent) = 0
```

- However, our statemachine is the one with the functions for doing this.
  - The solution is simple – have our state take in a reference to the state machine as well as our agent.

State Class

- Looking back at our state class, remember we want our State functions to have control over changing to different states.

- However, our statemachine is the one with the functions for doing this.
  - The solution is simple – have our state take in a reference to the state machine as well as our agent.

```
class State
   func Update(Agent agent, StateMachine sm) = 0
   func Init(Agent agent) = 0
   func Exit(Agent agent) = 0
```

Canberra Institute of Technology

aie SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

StateMachine class

- The update function is simple
  - it just calls update on the current state.
- The get functions are trivial as they just return their respective variables.
- The ChangeState function has to do a few things, though.
  - It must exit the current state
  - Initialize the new state
  - Move the current state into prevState
  - Set current state to the new state

```
func StateMachine::ChangeState
                 (Agent agent, State newState)

   currentState.Exit(agent)
   newState.Init(agent)
   prevState = currentState
   currentState = newState
```

Example State

- So lets make an example of a state from our agent.

- Here we're implementing the ChasePlayer state.
  - The goal of this state was to just seek towards the player
  - If we can't see the player we change to searching for it
  - If we get close to the player, we attack

```
class ChasePlayerState : State
   func Update(Agent agent, StateMachine sm)

      agent.seekToTarget()

      if ( !rayCast(agent, agent.seekTarget) )
         sm.ChangeState(agent, SearchForPlayer())

      if ( dist(agent.position, agent.seekTarget.position) < 5 )
         sm.ChangeState(agent, AttackPlayer())


   func Init(Agent agent)
       agent.seekTarget = player


   func Exit(Agent agent)
```

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

FSM drawbacks

- FSM’s are great for small systems.
  - However, as they get bigger and more complex, FSMs become difficult to manage.
  - Its easy to not have transitions set up correctly
  - Agents can get stuck in loops there’s not way to get out of.
  - We’ll soon look at some decision making structures that solve these problems

Questions?