AI for games

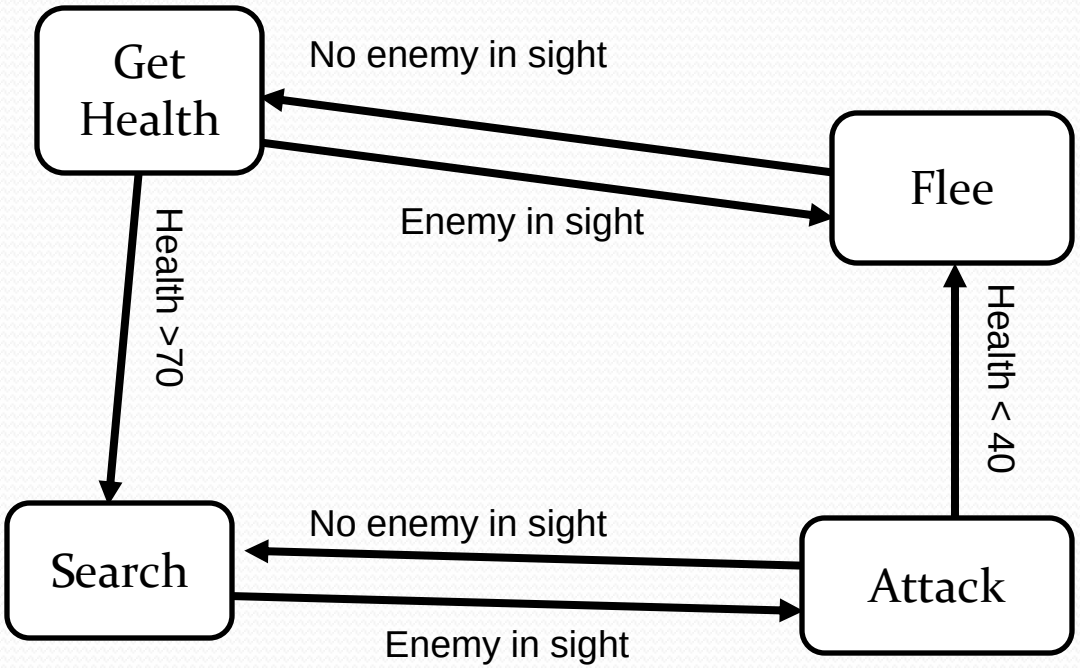
Lecture 5

Finite state machines

Organiser

- “Official” FSMs
- Game FSMs
  - Controlling the low-level behaviours
  - Switch statements
  - State class
  - State managers
- Messages

Organiser


Get Health
No enemy in sight
Flee
Enemy in sight
Health >70
Health < 40
Search
No enemy in sight
Attack
Enemy in sight

“Official” FSMs

- To a real computing scientist, a FSM is a function that maps an ordered sequence of input states to a corresponding sequence of input events.

Game FSMs

- Game programmers use what works.
- Mostly use a (slightly tweaked) version of the “state” design pattern, possibly with a “state machine” pattern thrown in for good measure.

Game FSMs

- The “ability” of a modern game AI is largely dependent on:
  - The complexity of the state machine.
  - Luck and patience in tinkering the rules of state changes.
  - The AI programmer being a sneaky bastard within each state to come up with clever ways of doing each state. (Which target? How to hide?)

Connecting to behaviours

- But first, how can a state ultimately control the entity?
- So far we have:
  - A collection of behaviours.
  - A pathfinding system.
- What handle do we have to control these?

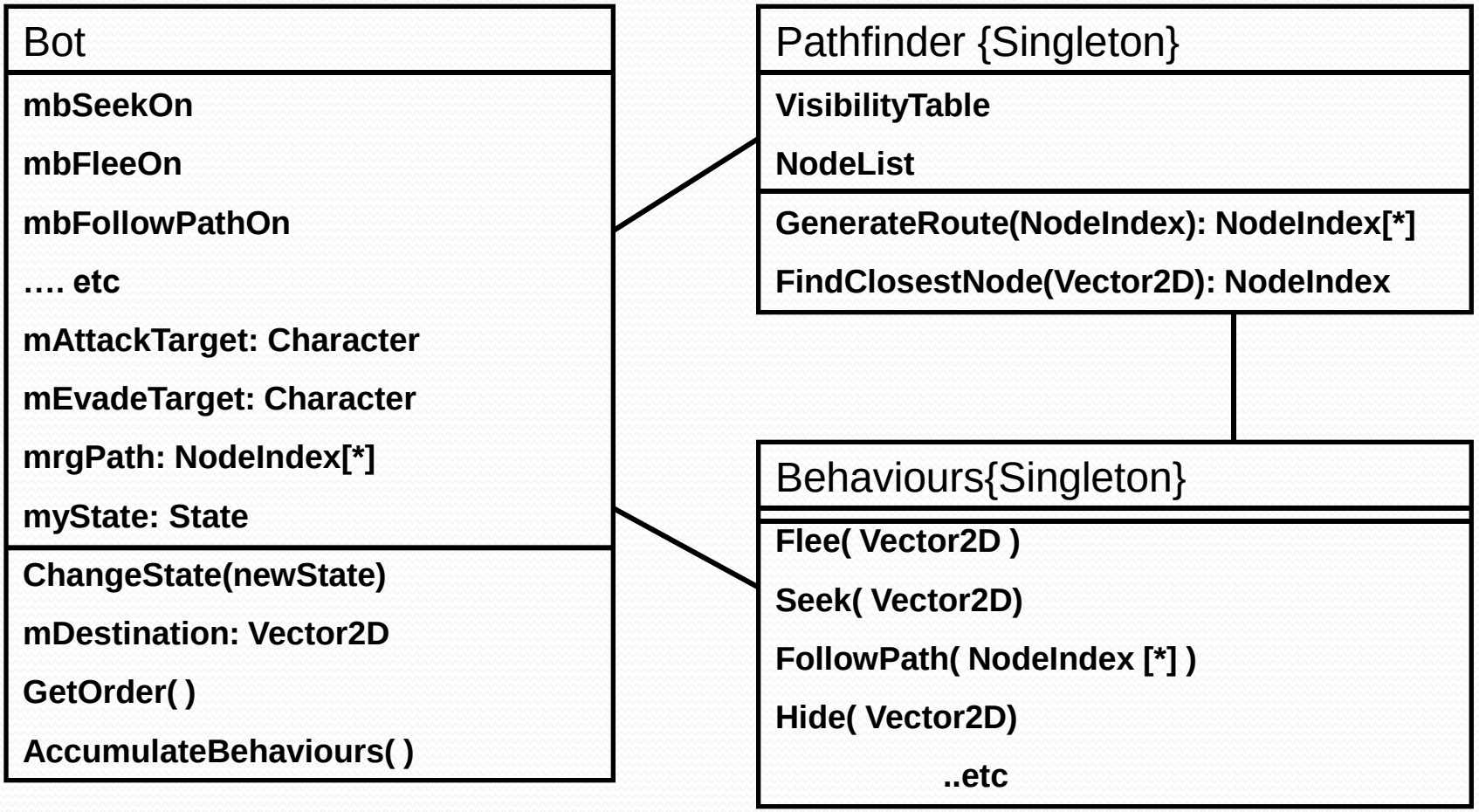
Connecting to behaviours

- A state can set several things:
  - Turning behaviours on and off, using bSeek, bAvoidWalls, bFlee, etc.
  - Specifying the target entity (seek, shoot at, etc)
  - Specifying the thing to “flee” or “hide” from.
  - Specifying a location (for “arrive at”).
  - Specifying a location (for “pathfind to”).

Connecting to behaviours

- Dozens of different architectures are arguable.
  - Make up your own.
- Let's start by doing everything wrong.

Connecting to behaviours



Behaviour changing

- Using this approach, each bot has a set of booleans and a few other variables to set.
- For example, on entering the FLEE state:

  mbFlee = true;

  mbAvoidWalls = true;
- … others to false

  mEvadeTarget = MostDangerousOpponent().mPosition;
- … etc

State management

- This means that most of the changes are associated with a CHANGE of state (a transition).
- During a state, all the state has to do is:
  - Check for a change of state.
  - Check for state-specific changes, like a change of target.
- Unfortunately, this turns out to be horribly ugly.

State management

```
enum State {FLEE, HUNT, FOLLOWPATH};
State myState;
Orders Bot ::GetOrders( )
{
    switch(myState)
    {
      case FLEE:
      // Flee checking code
      if(blah blah blah)
        ChangeState(HUNT);
      case HUNT:
      // Hunt specific code
      case GETHEALTHPACK:
      // Followpath specific code
    }
    AccumulateBehaviours();
};
```

State management

```
void ChangeState(State newState)
{
        switch(newState)
        {
                case FLEE:
                // Flee entry code
                case HUNT:
                // Hunt entry code
                case GETHEALTHPACK:
                // Followpath entry code
        }
        myState = newState;
}
```

State management

- Why is this horrible?

State management

- Remember this.....?

- The complexity of the state chart.
  - Fiddly to add new states.
- Luck and patience in tinkering the rules of state changes.
  - Keep having to recompile the entire thing.
- The AI programmer being sneaky.
  - Limited to programming within these booleans, etc.

The State Class

- We want:
  - To be able to add new states until the deadline. (Without invalidating the old ones.)
  - Freedom to code what we want within those states.
  - Encapsulation.

- *"Give me object-orientation or give me death!"*
  - *Patrick Henry 1774 (slightly amended)*

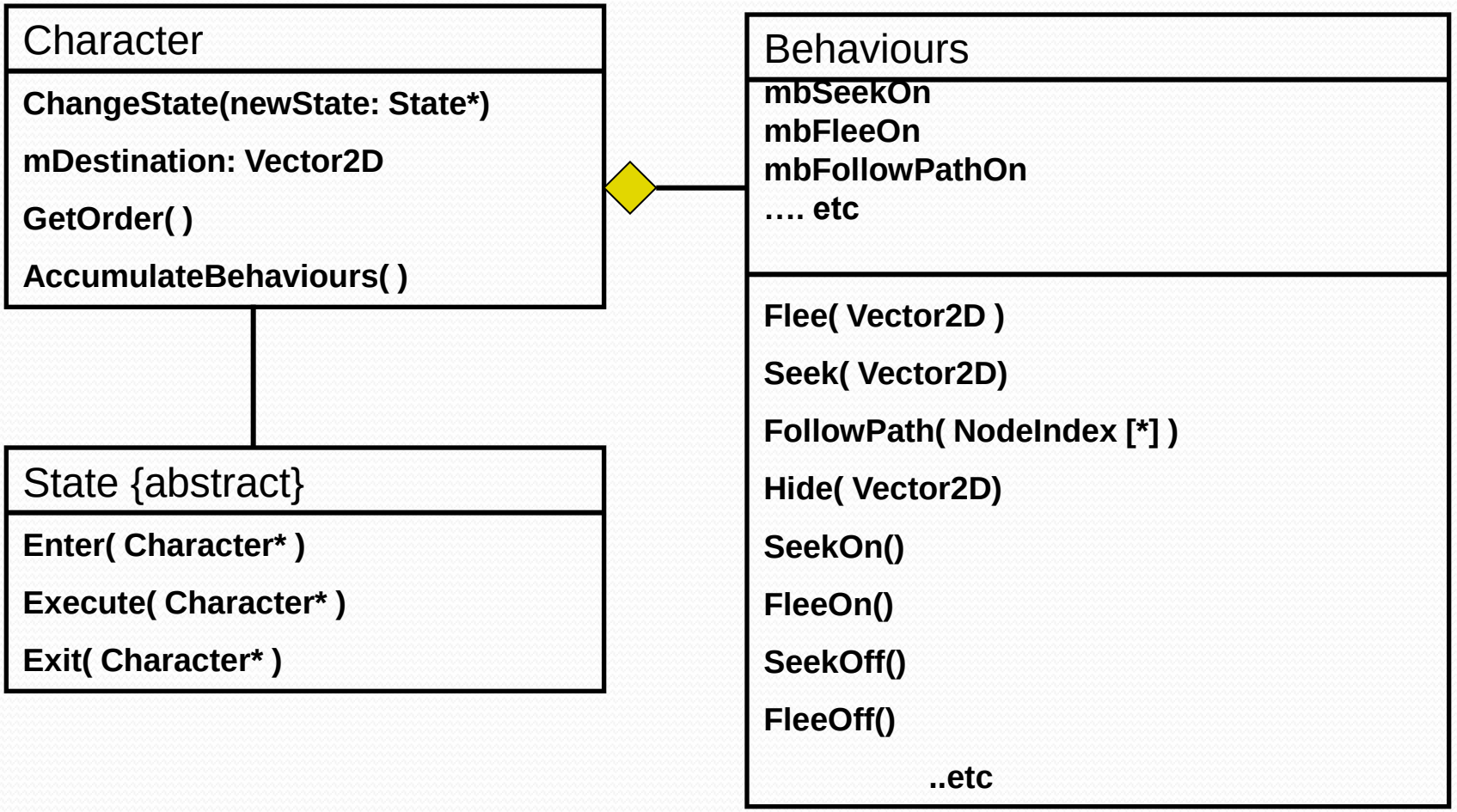
The State Class

- We are going to use the “state” design pattern.
- The Bot is connected to a state object that can be removed and replaced by another when the character wants to change state.

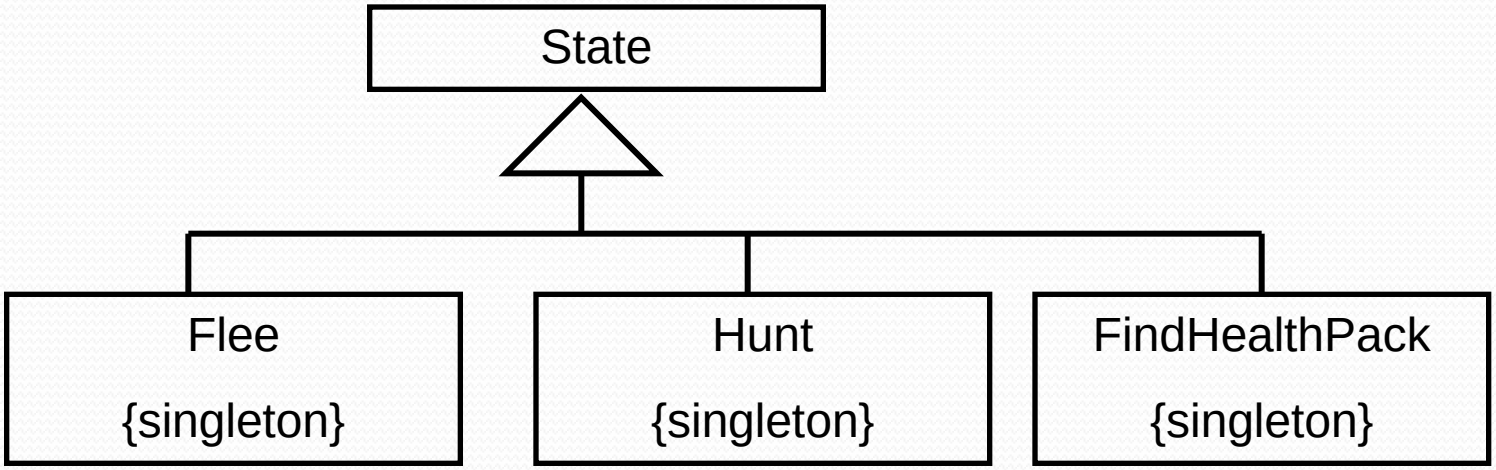
The State Class

- To avoid lots of real-time creation and destruction, the states will pre-exist, and will be effectively singletons.
- This means that each state cannot contain a memory, since Bots have to share it.
- So all these booleans have to be stored in the Bot?
- Why not put them in the Behaviours class.
  - It's just a list of functions at the moment anyway.

The State Class



State chart


State
Flee
{singleton}
Hunt
{singleton}
FindHealthPack
{singleton}

The Bot Class

```
Order CBot::GetOrder()
{
        if(pCurrentState)
                pCurrentState->Execute(this);
}
```

The Bot Class

```
void CBot::ChangeState(State* pNewState)
{
        if(pNewState)
        {
                pCurrentState->Exit(this);
                pCurrentState = pNewState;
                pCurrentState->Enter(this);
        }
}
```

- Some states (“check your six”) often want to return to a previous states when complete.

The Character Class

```
void Bot::ChangeState(State* pNewState)
{
        if(pNewState)
        {
                pCurrentState->Exit(this);
                pPreviousState=pCurrentState;
                pCurrentState = pNewState;
                pCurrentState->Enter(this);
        }
};
```

Singleton states

- Issues about the nature of singleton states.
  - Saves memory, etc.
  - Harder to do states within states, though.
- Also note problem of the state needing access to lots of the data in the ‘Bot.
  - Can it be made a friend class?
  - What’s the problem with that?

Sanity check

- We have a load of singleton states.
- CBots connect to the state they are currently in.
- Each state has a pointer back to the bot, so it can access stuff.
- The CBot has a function that allows it to change state.
  - This can be called by the state itself.

The State

```
template<class EntityType>
class State
{
private:
        // Why not put constructors (singleton!) here?
public:
        virtual void Enter(EntityType*)=0;
        virtual void Execute(EntityType*)=0;
        virtual void Exit(EntityType*)=0;
};
```

The State

- It may seem a bit pointless to template, as the actual states are pretty entity-specific, as they access lots of entity data.
- But the idea is to make this a “Component” that can be reused.
  - The actual states are game-specific, but the state framework is reusable.

Substates

```
class Flee: public State<CBot>
{
private:
        Flee();
        ~Flee();        // And other singleton stuff
        Flee* instance;
        CBot* pEnemy;
public:
        static Flee* GetInstance();
        void Enter(CBot* pBot);
        void Exit(CBot* pBot);
        void Execute(CBot* pBot);
};
```

Substates

```
void Flee::Enter(CBot* pBot)
{
   pEnemy = FindMostDangerousEnemy();
   pBot ->mBehaviours.mbFleeOn;
   pBot ->mBehaviours.mbArriveOff;
   // Etc
}

void Flee::Execute(CBot* pBot)
{
   pBot->mVelocity +=
        pBot->mBehaviours.AccumulateBehaviours();
   if(!Visible(pEnemy))
        pBot->ChangeState(FindHealthPack::GetInstance());
}
```

Substates

- Now one programmer can continue to work on this very basic flee state.
- Another can program the FindHealthPack state.
- And a third can work on developing new states, like FindBiggerGun.
  - New states can be added at will.
  - Or can they?

Substates

- Big problem is that once new states are added, some one has to tinker the old states so that they will actually get used.

- Not a bit problem, as long as the AI team talk to each other occasionally. Maybe even plan a bit……

Tricks and tips – Handy methods

- Some handy methods:
  - SelectBestTarget()
    - To allow on the fly update.
    - But make it reluctant to change target or may oscillate.
  - SelectMostDangerousEnemy()
    - Ditto

Tricks and tips – Handy methods

- SelectBestTarget()
  - Add up a “rating” for each enemy and return best.
  - Depends on:
    - Range
    - Aiming at me?
    - Aiming at colleague?
    - Easy to hit?
    - Dangerous?
    - Easy kill?
    - Close to strategic point?
    - Bonus for being current target.

Tricks and tips – Tumbleweed

- Tumbleweed is a state where you don't have a clue what is going on.
- Switch to it on error conditions you don't have time to write full code for.
  - Following path, but not apparently moving.
  - Escaping an enemy, but the enemy is now dead.
- Tumbleweed reappraises the situation from scratch and chooses the best state.

Tricks and tips – Reappraisal

- Many states should reappraise while executing.
  - Should I change the enemy I am running from?
  - Has my enemy been damaged? He may be weaker than me.
- Often, this sort of prioritisation code can be slow.
  - Don't have to do every frame!
  - How often depends on the individual state- compare "dodge" with "followpath".

Tricks and tips – Continuous appraisal state assignment

- An alternative architecture is to give each state an "Evaluate()" function that returns how important it is to do this task.
- One a state has finished (and at intervals while executing), each state gets evaluated to see the best thing the character could be doing.
- Each gives a score between 0 (don't do this) and 1 (absolutely must do this).

Tricks and tips – Continuous appraisal state assignment

- Flee .06
- FindHealthPack .45
- Attack .86
- Hunt .34
- Tumbleweed .0

- Again, give a bonus to the "current state" (depending on state, but especially for Attack.)
- Great for fuzzy logic, without defuzzification.

Tricks and tips – Continuous appraisal state assignment

- Advantages:
  - Can now add new states without changing previous ones.
- Disadvantages
  - Lots of tinkering.
  - May find some states never get used.

Summary

- The State class
- Changing states
- Using behaviours within the state class
- Some tricks and tips