Topic 13: Finite State Machines
State-Driven Agent Design

• Finite state machines, or FSMs as they are usually referred to, have for many years been the AI coder's instrument of choice to imbue a game agent with the illusion of intelligence.

• You will find FSMs of one kind or another in just about every game to hit the shelves since the early days of video games, and despite the increasing popularity of more esoteric agent architectures, they are going to be around for a long time to come. Here are just some of the reasons why:
  – They are quick and simple to code.
  – They are easy to debug.
  – They have little computational overhead.
  – They are intuitive.
  – They are flexible.
What is a FSM?

- A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.

- The idea behind a finite state machine, therefore, is to decompose an object's behavior into easily manageable "chunks" or states.
Games that use FSM

• The ghosts' behavior in Pac-Man
  – Evade state, which is the same for all ghosts, and
  – Each ghost has its own Chase state, the actions of which are implemented differently for each ghost.
  – The input of the player eating one of the power pills is the condition for the transition from Chase to Evade.
  – The input of a timer running down is the condition for the transition from Evade to Chase.

• Quake-style bots are implemented as finite state machines.

• Even the weapons in Quake implement their own mini finite state machines. For example, a rocket may implement
  – States: Move, TouchObject, and Die.
States typical in game environments
Good practice

• Visualising behaviour using the state diagram is a great way of understanding a specific behaviour.

• For your project, try to think of all possible states you want to have, as well as what rules to apply. The more planning, designing, and thinking you do beforehand, the less actual code you will have to write.

• On top of the core, the unit AI should also be able handle unit specific things, such as magician casting spells, and so force. It also should take care of unit animation and creating effects.
```
Implementation

switch(state)
{
    case STATE_IDLE:
    {
        // Idle Behaviour
        break;
    }
    case STATE_MOVING:
    {
        // Moving Behaviour
        break;
    }
    case STATE_DEAD:
    {
        // Stay Dead
        break;
    }
    case STATE_SEARCH:
    {
        // Get and select target within attack range
        break;
    }
    case STATE_ATTACK:
    {
        // Attack Selected Target
        break;
    }
    case STATE_GOTO_BUILD:
    {
        // Goto Building Behaviour
        break;
    }
    case STATE_BUILD:
    {
        // Build Building Behaviour
        break;
    }
    case STATE_IDLE:
    {
        animation = ANIM_STILL;
        enemies = EnemiesInRange();
        if(enemies.Size() > 0)
        {
            Target = BestTargetToAttack(enemies);
            state = STATE_ATTACK;
            result = true;
        }
        Heal();
        break;
    }
}
```
Different ways of implementing FSM

• In some cases switches (or if’s) like this work fine – particularly when no more states are expected.

• But, as more states and conditions are added, the switch-case structure ends up looking like spaghetti very quickly, making the program flow difficult to understand and creating a debugging nightmare.

• In addition, it's inflexible and difficult to extend beyond the scope of its original design, should that be desirable... and it most often is.

• Other ways:
  – State Transition Table
  – State Design Pattern
State Transition Tables

- This table can be queried by an agent at regular intervals, enabling it to make any necessary state transitions based on the stimulus it receives from the game environment.

- Each State can be modelled as a separate object or function external to the agent. This provides a clean and flexible architecture.
Embedded Rules using State Design Pattern

• A game entity, for example, a Troll, has a currentState, and also methods to ChangeState(), update() currentState based on external stimulus.

• In other words, state transition rules are embedded in the state itself.
Example Code in C++

class Troll
{
    /* ATTRIBUTES OMITTED */
    State* m_pCurrentState;

    public:
        /* INTERFACE TO ATTRIBUTES OMITTED */
    void Update()
    {
        m_pCurrentState->Execute(this);
    }

    void ChangeState(const State* pNewState)
    {
        delete m_pCurrentState;
        m_pCurrentState = pNewState;
    }
};

class State
{
    public:
        virtual void Execute (Troll* troll) = 0;
};

//--------------------------------------------State_RunAway
class State_RunAway : public State
{
    public:

        void Execute(Troll* troll)
        {
            if (troll->isSafe())
            {
                troll->ChangeState(new State_sleep());
            }
            else
            {
                troll->MoveAwayFromEnemy();
            }
        }
};

//--------------------------------------------State_sleep
class State_sleep : public State
{
    public:

        void Execute(Troll* troll)
        {
            if (troll->isThreatened())
            {
                troll->ChangeState(new State_RunAway());
            }
            else
            {
                troll->Snore();
            }
        }
};
class Troll
{
    State currentState;
    public void Update()
    {
        currentState.Execute(this);
    }
    public void ChangeState(State newState)
    {
        currentState = newState;
    }
}

class State
{
    public virtual void Execute(Troll troll)
    {
    }
}

class State_RunAway : State
{
    public void Execute(Troll troll)
    {
        if (troll.IsSafe())
            troll.ChangeState(new State_Sleep());
        else
            troll.MoveAwayFromEnemy();
    }
}

class State_Sleep() : State
{
    public void Execute(Troll troll)
    {
        if (troll.isThreatened())
            troll.ChangeState(new State_RunAway());
        else
            troll.Snore();
    }
}
A more involved example

• As a practical example of how to create agents that utilize finite state machines, we are going to look at a game environment where agents inhabit an Old West-style gold mining town named West World. Initially there will only be one inhabitant — a gold miner named Miner Bob.

• You may have to imagine the tumbleweeds, creaking mine props, and desert dust blowing in your eyes because West World is implemented as a simple text-based console application. Any state changes or output from state actions will be sent as text to the console window.

• *Hold press - there’s a brand new graphical NeoAxis version. (Some imagination may still be required.)*
Miner’s State Transition

Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 3
Miner Bob: Leavin' the bank
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Boy, ah sure is thirsty! Walkin' to the saloon
Miner Bob: That's mighty fine sippin' liquor
Miner Bob: Leavin' the saloon, feelin' good
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 4
Miner Bob: Leavin' the bank
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Boy, ah sure is thirsty! Walkin' to the saloon
Miner Bob: That's mighty fine sippin' liquor
Miner Bob: Leavin' the saloon, feelin' good
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 5
Miner Bob: Woooohoo! Rich enough for now. Back home to mah li'l lady
Miner Bob: Leavin' the bank
Miner Bob: Walkin' home
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: What a God-darn fantastic nap! Time to find more gold
State Design Pattern

**BaseGameEntity**
- Update(): void

**Miner**
- Update(): void
- GetFSM(): StateMachine<Miner>*
- Thirst += 1;
  - StateMachine->Update()

**StateMachine<Miner>**
- SetGlobalState(State<Miner>*): void
- SetCurrentState(State<Miner>*): void
- Update(): void
- ChangeState(State<Miner>*): void
- RevertToPreviousState(): void

**State<Miner>**
- Enter(Miner*): void
- Execute(Miner*): void
- Exit(Miner*): void
  - Instance(): this

**MinerGlobalState**
- Enter(Miner*): void
- Execute(Miner*): void
- Exit(Miner*): void
  - Instance(): this

**EnterMineAndDigForNugget**
- Enter(Miner*): void
- Execute(Miner*): void
- Exit(Miner*): void
  - Instance(): this

**VisitBankAndDepositGold**
- Enter(Miner*): void
- Execute(Miner*): void
- Exit(Miner*): void
  - Instance(): this

**GoHomeAndSleepTillRestored**
- Enter(Miner*): void
- Execute(Miner*): void
- Exit(Miner*): void
  - Instance(): this

**QuenchThirst**
- Enter(Miner*): void
- Execute(Miner*): void
- Exit(Miner*): void
  - Instance(): this
Coding a general state design pattern

- Generics (as in C#, Java) can capture a general pattern
  - State generic class
  - StateMachine generic class

- State-design pattern
  - State Machine keeps track of previousState, globalState and currentState and responsible for
    - state transition, executing the current state and reverting back to the previous state
  - A game entity knows its current state and owns a finite state machine that kicks start with the current state
  - A set of game entity specific states – looks after behaviours inside that state and conditions for state transition
Singleton Design Pattern

• Often it's useful to guarantee that an object is only instantiated once and/or that it is globally accessible.

• For example, in game designs that have environments consisting of many different entity types — players, monsters, projectiles, plant pots, etc. — there is usually a "manager" object that handles the creation, deletion, and management of such objects.

• There are many ways of doing this, but very often through a static Instance() method.
Wikipedia on Code Refactoring

• In **software engineering**, "refactoring" source code means modifying it without changing its behavior, and is sometimes informally referred to as "cleaning it up".

• Refactoring neither fixes **bugs** nor adds new functionality, though it might precede either activity. Rather it improves the **understandability** of the code and changes its internal **structure and design**, and removes **dead code**, to make it easier to comprehend, more maintainable and amenable to change.

• Refactoring is usually motivated by the difficulty of adding new functionality to a program or fixing a bug in it.
Global State

• More often than not, you will end up with code that is duplicated in every state. For example, in the game *The Sims* by Electronic Arts, a Sim may feel the urge of nature come upon it and have to visit the bathroom to relieve itself. This urge may occur in any state and at any time.

• Given the current design, three options:
  – duplicate conditional logic to be added to every one of his states, or alternatively
  – placed into the Update function of the Miner; it's better to
  – create a *global state* that is called every time the FSM is updated. That way, all the logic for the FSM is contained within the states and not in the agent class that owns the FSM.
FSMs via yield

• Yet another implementation technique is to use the technique of coroutines via “yield”, with mutually recursive methods for the states.

• This has the advantage that it scales naturally to complicated ways of combining behaviours beyond the FSM model.

```csharp
public class TrollYAI : YieldAI
{
    public IEnumerable<Object> Start()
    {
        return Sleep();
    }

    IEnumerable<Object> RunAway()
    {
        while (!this.IsSafe())
        {
            MoveAwayFromEnemy();
            yield return null;
        }
        yield return Sleep();
    }

    IEnumerable<Object> Sleep()
    {
        while (!IsThreatened())
        {
            Snore();
            yield return null;
        }
        yield return RunAway();
    }
}
```
public class YieldAI
{
    protected IEnumerator<Object> decisions;
    public YieldAI()
    {
        decisions = Start().GetEnumerator();
    }

    protected virtual IEnumerable<Object> Start();

    public void OnTick()
    {
        if (decisions.MoveNext())
        {
            var m = decisions.Current as IEnumerator<Object>;
            if (m != null)
            {
                decisions = m;
            }
        }
    }
}
What’s in NeoAxis

• GameCharacterAI Class
  – Two sets of three states
    • Attack/ForceAttack/NoAttack
    • Move/ForceMove/NoMove
  – Attacking and moving can be combined.
  – Target of Attack/Move can be a position or a dynamic entity.
  – OnTick method makes updates.
  – Works closely with GameCharacter.
• You may want to inherit directly from AI if you have other behaviour, and perhaps copy some parts.