# Worksheet for Week 2 – Finite State Machines

## Overview

### Objectives:

1. Design a basic Finite State Machine
2. Build an FSM using switch statements
3. Build an FSM with transition functions
4. Build a class-based FSM

### Bonus Objectives

1. Exploratory work with the FSM\_Unity\_Example.zip from MyLearningSpace

## Preliminary

Primarily this tutorial work will require the use of a C++ IDE. Visual Studio is the recommended software to use although any C++ development environment should be sufficient.

There is a Visual Studio Project on MyLearningSpace that you can download and use for this week’s work. A link to this will be available on MyLearningSpace.

The bonus objectives will require Unity (**version 8+**). Older versions may work but I cannot aid with import errors if you use an older version. Unity will require you to create a Unity Account as well.

If you need help setting up the software, we have a tool called AppsAnywhere that allows you access to some of the software we use. You can use this to install some of the software we use. <https://myapps.abertay.ac.uk/>

## Getting Started

Use the link in MyLearningSpace to access the project that contains the *Finite State Machine* assignment. Download the assignment onto your machine. Open the Visual Studio Project.

If you chose to use a different IDE or compiler setup, then copy the below code into a file and work from that.



## Design a Basic Finite State Machine

Using the lecture slides as a guide, create the states, transitions and overall function of the following state machines. You can do this on a computer (such as a PowerPoint slide) or you can do it with pen and paper if you wish.

1. A sports spectator is to have the following behaviours:

* By default they sit and watch the game
* If their team scores, they stand and cheer
* If the opposing team scores, they stand and boo
* After 15 seconds standing, for any reason, they sit down again
* If they are hungry at any time, they go to the food stall
* If they have food they sit and eat it and do not stand for any reason
* It takes five minutes to eat food, after which they no longer have food
* The spectator is never hungry while they have food

Design a finite state machine that will demonstrate the behaviour described above, briefly describe the FSM used in your design.

2. Using a directed graph design a simple, single level *Finite State Machine* that could be used to create the following ‘emotional model’ for an NPC.

* Default state is CALM
* If NPC hears a noise at any time and there is no one else about then they become ALERT
* If the noise stops and nothing else happens then the NPC becomes CALM
* If NPC sees another character at any time they become AGITATED
* If other character appears to be friendly then they become CALM
* If the other character starts to act aggressively then they are SURPRISED
* If the other character is friendly and cracks a joke then they become HAPPY
* If the other character is aggressive and attacks they become ANGRY
* If the other character moves away then they become CALM

## Building an FSM

### Switch Statements

Choose one of the examples from the previous task and implement it in a C++ program. It does not require visuals, it just needs to print messages onto the screen.

1. Create an *enum* that will contain the different states and populate it with each available state you have designed.
2. Create an instance of the *enum* at the beginning of your program. This will be used to determine which state you are in.
3. Add a switch statement to the do-while loop.
4. Add each *enum* value as a condition in the switch statement
5. For each member of the switch statement, add in the appropriate code so that it acts in accordance with your design.
6. Add in delays in your code when necessary
7. Use keyboard inputs to act as events if you need to

For example, when you press the N key, it can trigger a ‘Noise’ event

Notes:

* Make sure you initialise the *enum* with a starting state
* The easiest way to implement and manage all the state code is to have a separate function for each
* The project contains both basic delay code (*sleep\_for*) and keyboard input code *(\_kbhit* and *\_getch*). Feel free to repurpose this code as you see fit.
* The *\_kbhit* and *\_getch* commands will not work on all compilers. If this is the case, you can use whatever form of input handling you like.

### Transition Functions

1. If you have not already done so, create a function that handles the transition between states. The following example code may be useful:



1. Add a pre-switch function that prints out the current state and what it will turn to.
2. Create a similar post-switch function

Optional Tasks

If you want to, have separate functions for each pre and post function for every possible transition. You can do this in a number of ways. Either create them all manually or store them as *std::function* in a map and access them using the current state.

### Class Based FSM

We are now going to rework the Finite State Machine so it uses classes rather than a switch statement.

1. Start by adding the following class into your code:



1. Create a new class for each state of your program, making sure it inherits from BaseState.
2. In each child class, overwrite the run with code specific to that state
3. Create a single instance of each class in your program (you can do this by hardcoding them in or by using an std::map)
4. Create a pointer to a BaseState that will function as the current state the program runs.
5. Assign the pointer to a starting state.
6. Edit the transition function so it can swap the pointer from its current target to the new desired state
7. Replace the switch statement from previous tasks with something along the following lines (this is just pseudocode so it won’t work without some tweaking):



## Bonus Objectives

Download the FSM Unity Example from MyLearningSpace and import into Unity. You will need Unity version 2018.1 (or higher), it may import into lower version of Unity but there is no guarantee.

The following graph indicates the structure of the AI in the game.

A close up of a logo

Description automatically generated

1. Run the game and view the AI running
   1. Write down what the events are the trigger a state change
2. Add a new state to the game where if the AI is chasing the player and the player is above them, try and get the AI to jump as well. Return to either chase or patrol state after the jump is complete.

### Open Ended Work

Extend the AI in any way that takes your fancy. Integrate pathfinding so it can follow the player across the platforms, add attack methods and states, add a stunned state if the player collides with the bot’s head etc.

Invent something creative that extends the current Finite State Machine.